THE EXPERT'S VOICE® IN OPEN SOURCE

Dive Into Python

All you need to know to get off the ground and start to fly with Python programming!

Mark Pilgrim

Apress®

目錄

Dive Into Python

版权信息

审校 (5.4b)：2007 年 6 月—9 月

译文版 (5.4)：2005 年 12 月—2006 年 4 月 (update-060425)

英文原版 (5.4)：2004 年 5 月 20 日

本书存放在 http://diveintopython.org/ (英文原版) 和 http://www.woodpecker.org.cn/diveintopython(中文版)。如果你是从别的地方看到它的，可能看到的不是最新版本。

本译本由 Zoom.Quiet 负责项目管理。感谢啄木鸟社区提供 SVN 项目空间和 Wiki 协作空间。

本译本由 啄木鸟/CPUG 的 obp 团队完成。可以在修订历史中找到一个翻译和修订人员的清单。如果您对当前版本的 Dive Into Python 中文版有任何意见和建议，可以到本书的 Wiki 协作空间中留下你的评论。

第 1 章 安装 Python

欢迎来到 Python 世界，让我们开始吧。在本章中，将学习适合您的 Python 安装。

1.1. 哪一种 Python 适合您？

学习 Python 的第一件事就是安装，不是吗？

如果您在公网的服务器上有个用户账号，那么您的 ISP 或许已经安装了 Python。 大多数 Linux 发行版在默认安装的情况下就已经提供了 Python。 虽然您可能希望在苹果机上安装一个拥有类 Mac 的图形操作界面，但在 Mac OS X 10.2 或更高的版本上已经包含了一个 Python 的命令行版本。

Windows 环境默认不提供任何版本的 Python，但是不要担心！本章将提供几种 Windows 环境下安装 Python 的方法。

正像您所看到的，Python 可以运行于很多操作系统平台。包括 Windows、Mac OS、Mac OS X、所有免费的类 UNIX 变种 (如 Linux)。也有运行于 Sun Solaris、AS/400、Amiga、OS/2、BeOS 的版本，甚至是您从来没听说过的其他操作系统平台。

有太多的平台可以运行 Python 了。在一种平台下编写的 Python 程序稍作修改，就可以运行于任何 其他支持的平台。例如，我通常在 Windows 平台上开发 Python 程序，然后适当配置后使之能在 Linux 平台上运行。

回到开始的问题，“哪一种 Python 适合您？” 回答是：哪一个已经安装在您计算机上均可。

1.2. Windows 上的 Python

在 Windows 上，安装 Python 有两种选择。

ActiveState 制作的 ActivePython 是专门针对 Windows 的 Python 套件，它包含了一个完整的 Python 发布、一个适用于 Python 编程的 IDE 以及一些 Python 的 Windows 扩展，提供了全部的访问 Windows APIs 的服务，以及 Windows 注册表的注册信息。

虽然 ActivePython 不是开源软件，但它可以自由下载。ActivePython 是我学习 Python 时使用过的 IDE。除非有别的原因，我建议您使用它。可能的一个原因是：ActiveState 通常要在新的 Python 版本发布几个月以后才更新它的安装程序。如果您就需要 Python 的最新版本，并且 ActivePython 仍然落后于最新版本的话，您应该直接跳到在 Windows 上安装 Python 的第二种选项。

第二种选择是使用由 Python 发布的 “官方” Python 安装程序。她是可自由下载的开源软件，并且您总是可以获得当前 Python 的最新版本。

过程 1.1. 选项 1：安装 ActivePython

下面描述 ActivePython 的安装过程：

1. 从 http://www.activestate.com/Products/ActivePython/ 下载 ActivePython 。
2. 如果您正在使用 Windows 95、Windows 98 或 Windows ME，还需要在安装 ActivePython 之前下载并安装Windows Installer 2.0 。
3. 双击安装程序 `ActivePython-2.2.2-224-win32-ix86.msi` 。
4. 按照安装程序的提示信息一步步地执行。
5. 如果磁盘空间不足，您可以执行定制安装，不选文档，但是笔者不建议您这样做，除非您实在是挤不出14M空间来。
6. 在安装完后之后，关闭安装程序，打开 开始->程序->ActiveState ActivePython 2.2->PythonWin IDE。您将看到类似如下的信息：

```
PythonWin 2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)] on win32.
Portions Copyright 1994-2001 Mark Hammond (mhammond@skippinet.com.au) -
see 'Help/About PythonWin' for further copyright information.
>>>
```

过程 1.2. 选项 2：安装来自 Python.org 的 Python

1. 从 http://www.python.org/ftp/python/ 选择最新的 Python Windows 安装程序，下载 `.exe` 安装文件。
2. 双击安装程序 `Python-2.xxx.yyy.exe` 。文件名依赖于您所下载的 Python 安装程序文件。
3. 按照安装程序的提示信息一步步地执行。
4. 如果磁盘空间不足，可以取消 HTMLHelp 文件、实用脚本 (`Tools/`)、和/或测试套件 (`Lib/test/`)。
5. 如果您没有机器的管理员权限，您可以选择 Advanced Options，然后选择 Non-Admin Install。这只会对登记注册表和开始菜单中创建的快捷方式有影响。
6. 在安装完成之后，关闭安装程序，打开 开始->程序->Python 2.3->IDLE (Python GUI)。您将看到类似如下的信息：

```
Python 2.3.2 (#49, Oct  2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************
IDLE 1.0
>>>
```

1.3. Mac OS X 上的 Python

在 Mac OS X 上，对于安装 Python 有两种选择：安装或不安装。您可能想要安装它。

Mac OS X 10.2 及其后续版本已经预装了一个 Python 的命令行版本。如果您习惯使用命令行，那么您可以使用它学完本书的三分之一。然而，预安装的版本不带 XML 解析器，所以当您学到 XML 的章节时，您会需要安装完整版。

您还可以安装优于预装版本的最新的包含图形界面 Shell 的完整版本。

过程 1.3. 在 Mac OS X 上运行预装版本的 Python

使用预装的 Python 版本的步骤：

1. 打开 `/Applications` 文件夹。
2. 打开 `Utilities` 文件夹。
3. 双击 `Terminal` 打开一个终端进入命令行窗口。
4. 在提示符下键入 **`python`** 。

试验:

```
Welcome to Darwin!
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you%
```

过程 1.4. 在 Mac OS X 上安装最新版的 Python

下面介绍下载并安装 Python 最新版本的过程:

1. 从 http://homepages.cwi.nl/~jack/macpython/download.html 下载 `MacPython-OSX` 磁盘镜像 。
2. 下载完毕，双击 `MacPython-OSX-2.3-1.dmg` 将磁盘镜像挂载到桌面。
3. 双击安装程序 `MacPython-OSX.pkg` .
4. 安装程序将提示要求您的管理员用户名和口令。
5. 按照安装程序的提示一步步执行。

6. 安装完毕后，关闭安装程序，打开 `/Applications` 文件夹。
7. 打开 `MacPython-2.3` 文件夹。
8. 双击 `PythonIDE` 来运行 Python 。

MacPython IDE 将显示一个弹出屏幕界面将您带进交互 shell。如果交互shell没有出现，选择 Window->Python Interactive (****Cmd**-0**)。您将看到类似如下的信息:

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

请注意，安装完最新版本后，预装版本仍然存在。如果您从命令行运行脚本，那您需要知道正在使用的是哪一个版本的 Python 。

例 1.1. 两个 Python 版本

```
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you% /usr/local/bin/python
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you%
```

1.4. Mac OS 9 上的 Python

Mac OS 9 上没有预装任何版本的 Python，安装相对简单，只有一种选择。

下面介绍在 Mac OS 9 上安装 Python 的过程:

1. 从 http://homepages.cwi.nl/~jack/macpython/download.html 下载 `MacPython23full.bin` 。
2. 如果浏览器不能自动解压文件，那么双击 `MacPython23full.bin` 用 Stuffit Expander 解压。
3. 双击安装程序 `MacPython23full` 。
4. 按照安装程序的提示一步步执行。
5. 安装完毕后，关闭安装程序，打开 `/Applications` 文件夹。
6. 打开 `MacPython-OS9 2.3` 文件夹。
7. 双击 `PythonIDE` 来运行 Python 。

MacPython IDE 将显示一个弹出屏幕界面将您带进交互 shell。如果交互shell没有出现，选择 Window->Python Interactive (**Cmd-0**)。您将看到类似如下的信息:

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

1.5. RedHat Linux 上的 Python

在类 UNIX 的操作系统 (如 Linux) 上安装二进制包很容易。预编译好的二进制包对大多数 Linux 发行版是可用的。或者您可以通过源码进行编译。

在 http://www.python.org/ftp/python/ 选择列出的最新的版本号, 然后选择 其中的 `rpms/` 目录下载最新的 Python RPM 包。 使用 **rpm** 命令进行安装，操作如下所示:

例 1.2. 在 RedHat Linux 9 上安装

```
localhost:~$ su -
Password: [enter your root password]
[root@localhost root]# wget http://python.org/ftp/python/2.3/rpms/redhat-9/python2.3-2.3-
Resolving python.org... done.
Connecting to python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7,495,111 [application/octet-stream]
...
[root@localhost root]# rpm -Uvh python2.3-2.3-5pydotorg.i386.rpm
Preparing...                ########################################### [100%]
   1:python2.3              ########################################### [100%]
[root@localhost root]# python
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to exit]
[root@localhost root]# python2.3
Python 2.3 (#1, Sep 12 2003, 10:53:56)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to exit]
[root@localhost root]# which python2.3
/usr/bin/python2.3
```

[1]	仅仅键入 `python` 运行的是老版本的 Python ——它是缺省安装的版本。它不是我们想要的。
[2]	截止到笔者写作时，新的版本是 `python2.3` 。您可能会需要修改示例脚本的第一行的路径指向新版本。
[3]	这是我们刚安装的 Python 新版本的全路径。在 `#!` 行中 (每个脚本的第一行) 使用它来确保脚本运行在最新版的 Python 下，并且确保敲入的是 `python2.3` 进入交互 shell。

1.6. Debian GNU/Linux 上的 Python

如果您运行在 Debian GNU/Linux 上，安装 Python 需要使用 **apt** 命令。

例 1.3. 在 Debian GNU/Linux 上安装

```
localhost:~$ su -
Password: [enter your root password]
localhost:~# apt-get install python
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  python2.3
Suggested packages:
  python-tk python2.3-doc
The following NEW packages will be installed:
  python python2.3
0 upgraded, 2 newly installed, 0 to remove and 3 not upgraded.
Need to get 0B/2880kB of archives.
After unpacking 9351kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Selecting previously deselected package python2.3.
(Reading database ... 22848 files and directories currently installed.)
Unpacking python2.3 (from .../python2.3_2.3.1-1_i386.deb) ...
Selecting previously deselected package python.
Unpacking python (from .../python_2.3.1-1_all.deb) ...
Setting up python (2.3.1-1) ...
Setting up python2.3 (2.3.1-1) ...
Compiling python modules in /usr/lib/python2.3 ...
Compiling optimized python modules in /usr/lib/python2.3 ...
localhost:~# exit
logout
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [press Ctrl+D to exit]
```

1.7. 从源代码安装 Python

如果您宁愿从源码创建，可以从 http://www.python.org/ftp/python/下载 Python 的源代码。选择最新的版本，下载 `.tgz` 文件，执行通常的 **`configure`** , **`make`** , **`make install`** 步骤。

例 1.4. 从源代码安装

```
localhost:~$ su -
Password: [enter your root password]
localhost:~# wget http://www.python.org/ftp/python/2.3/Python-2.3.tgz
Resolving www.python.org... done.
Connecting to www.python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8,436,880 [application/x-tar]
...
localhost:~# tar xfz Python-2.3.tgz
localhost:~# cd Python-2.3
localhost:~/Python-2.3# ./configure
checking MACHDEP... linux2
checking EXTRAPLATDIR...
checking for --without-gcc... no
...
localhost:~/Python-2.3# make
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Modules/python.o Modules/python.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/acceler.o Parser/acceler.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/grammar1.o Parser/grammar1.c
...
localhost:~/Python-2.3# make install
/usr/bin/install -c python /usr/local/bin/python2.3
...
localhost:~/Python-2.3# exit
logout
localhost:~$ which python
/usr/local/bin/python
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
localhost:~$
```

1.8. 使用 Python 的交互 Shell

既然我们已经安装了 Python，那么我们运行的这个交互shell是什么东西呢？

Python 扮演着两种角色。首先它是一个脚本解释器，可以从命令行运行脚本，也可以在脚本上双击，像运行其他应用程序一样。它还是一个交互 shell，可以执行任意的语句和表达式。这一点对调试、快速组建和测试相当有用。我甚至知道一些人把 Python 的交互 shell 当作计算器来使用！

在您的计算机平台上启动 Python 的交互 shell，接下来让我们尝试着做些操作：

例 1.5. 初次使用交互 Shell

```
>>> 1 + 1
2
>>> print 'hello world'
hello world
>>> x = 1
>>> y = 2
>>> x + y
3
```

[1]	Python 的交互 shell 可以计算任意的 Python 表达式，包括任何基本的数学表达式。
[2]	交互 shell 可以执行任意的 Python 语句，包括 **print** 语句。
[3]	也可以给变量赋值，并且变量值在 shell 打开时一直有效 (一旦关毕交互 Sheel，变量值将丢失)。

1.9. 小结

您现在应该已经安装了一个可以工作的 Python 版本了。

根据您的运行平台，您可能安装有不止一个 Python 版本。那样的话，您需要知道 Python 的路径。若在命令行简单地键入 **python** 没有运行您想使用的 Python 版本，则需要输入想要的版本的全路径。

最后祝贺您，欢迎来到 Python 世界。

第 2 章 第一个 Python 程序

大家都很清楚，其他书籍是如何一步步从编程基础讲述到构建完整的可运行程序的，但还是让我们跳过这个部分吧！

2.1. 概览

这是一个完整的、可执行的 Python 程序。

它可能对您来说根本无法理解。别着急，我们将逐行地进行剖析。不过首先把代码通读一遍，看一看是否有些可以理解的内容。

例 2.1. `odbchelper.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
 def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.
    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

现在运行一下这个程序，看一看结果是什么。

> 提示
> 在 Windows 的 ActivePython IDE 中，可以选择 File->Run... (**Ctrl-R**) 来运行 Python 程序。输出结果将显示在交互窗口中。
>
> 提示
> 在 Mac OS 的 Python IDE 中，可以选择 Python->Run window... (**Cmd-R**) 来运行 Python 程序，但首先要设置一个重要的选项。在 IDE 中打开 `.py` 模块，点击窗口右上角的黑色三角，弹出这个模块的选项菜单，然后将 Run as **main** 选中。 这个设置是同模块一同保存的，所以对于每个模块您都需要这样做。
>
> 提示
> 在 UNIX 兼容的操作系统中 (包括 Mac OS X)，可以通过命令行：`python` **odbchelper.py**`` 运行模块。

`odbchelper.py` 的输出结果：

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

2.2. 函数声明

与其它大多数语言一样 Python 有函数，但是它没有像 C++ 一样的独立的头文件；或者像 Pascal 一样的分离的 `interface` / `implementation` 段。在需要函数时，像下面这样声明即可：

```
def buildConnectionString(params):
```

首先，函数声明以关键字 `def` 开始，接着为函数名，再往后为参数，参数放在小括号里。多个参数之间 (这里没有演示)用逗号分隔。

其次，函数没有定义返回的数据类型。Python 不需要指定返回值的数据类型；甚至不需要指定是否有返回值。实际上，每个 Python 函数都返回一个值；如果函数执行过 `return` 语句，它将返回指定的值，否则将返回 `None` (Python 的空值)。

> 注意
> 在 Visual Basic 中，函数 (有返回值) 以 `function` 开始，而子程序 (无返回值) 以 `sub` 开始。在 Python 中没有子程序。只有函数，所有的函数都有返回值 (尽管可能为 `None`)，并且所有的函数都以 `def` 开始。

最后需要指出的是，在 Python 中参数， `params` 不需要指定数据类型。Python 会判定一个变量是什么类型，并在内部将其记录下来。

> 注意
> 在 Java、C++ 和其他静态类型语言中，必须要指定函数返回值和每个函数参数的数据类型。在 Python 中，永远也不需要明确指定任何东西的数据类型。Python 会根据赋给它的值在内部将其数据类型记录下来。

2.2.1. Python 和其他编程语言数据类型的比较

一位博学的读者发给我 Python 如何与其它编程语言的比较的解释：

静态类型语言

一种在编译期间就确定数据类型的语言。大多数静态类型语言是通过要求在使用任一变量之前声明其数据类型来保证这一点的。Java 和 C 是静态类型语言。

动态类型语言

一种在运行期间才去确定数据类型的语言，与静态类型相反。VBScript 和 Python 是动态类型的，因为它们确定一个变量的类型是在您第一次给它赋值的时候。

强类型语言

一种总是强制类型定义的语言。Java 和 Python 是强制类型定义的。您有一个整数，如果不明确地进行转换 ，不能将把它当成一个字符串。

弱类型语言

一种类型可以被忽略的语言，与强类型相反。VBScript 是弱类型的。在 VBScript 中，您可以将字符串 `'12'` 和整数 `3` 进行连接得到字符串`'123'`，然后可以把它看成整数 `123` ，所有这些都不需要任何的显示转换。

所以说 Python 既是动态类型语言 (因为它不使用显示数据类型声明)，又是强类型语言 (因为只要一个变量获得了一个数据类型，它实际上就一直是这个类型了)。

2.3. 文档化函数

可以通过给出一个 `doc string` (文档字符串) 来文档化一个 Python 函数。

例 2.2. 定义 `buildConnectionString` 函数的 `doc string`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.
    Returns string."""
```

三重引号表示一个多行字符串。在开始与结束引号间的所有东西都被视为单个字符串的一部分，包括硬回车和其它的引号字符。您可以在任何地方使用它们，但是您可能会发现，它们经常被用于定义 `doc string` 。

> 注意
> 三重引号也是一种定义既包含单引号又包含双引号的字符串的简单方法，就像 Perl 中的 `qq/.../` 。

在三重引号中的任何东西都是这个函数的 `doc string` ，它们用来说明函数可以做什么。如果存在 `doc string` ，它必须是一个函数要定义的第一个内容 (也就是说，在冒号后面的第一个内容)。在技术上不要求给出函数的 `doc string` ，但是您应该这样做。我相信在您上过的每一种编程课上都听到过这一点，但是 Python 带给您一些额外的动机： `doc string` 在运行时可作为函数的属性。

> 注意
> 许多 Python IDE 使用 `doc string` 来提供上下文敏感的文档信息，所以当键入一个函数名时，它的 `doc string` 显示为一个工具提示。这一点可以说非常有用，但是它的好坏取决于您书写的 `doc string` 的好坏。

进一步阅读

- PEP 257 定义了 `doc string` 规范。
- *Python Style Guide* 讨论了如何编写一个好的 `doc string` 。
- *Python Tutorial* 讨论了在 `doc string` 中如何使用空白。

2.4. 万物皆对象

也许您没在意，我刚才的意思是 Python 函数有属性，并且这些属性在运行时是可用的。

在 Python 中，函数同其它东西一样也是对象。

打开您习惯使用的 Python IDE 执行如下的操作：

例 2.3. 访问 `buildConnectionString` 函数的 `doc string`

```
>>> import odbchelper
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> print odbchelper.buildConnectionString(params)
server=mpilgrim;uid=sa;database=master;pwd=secret
>>> print odbchelper.buildConnectionString.__doc__
Build a connection string from a dictionary
Returns string.
```

[1]	第一行将 `odbchelper` 程序作为模块导入。模块是指一个可以交互使用，或者从另一 Python 程序访问的代码段。(您在 第 4 章 将会看到多模块 Python 程序的许多例子。) 只要导入了一个模块，就可以引用它的任何公共的函数、类或属性。模块可以通过这种方法来使用其它模块的功能，您也可以在 IDE 中这样做。这是一个很重要的概念，在后面我们将谈得更多。
[2]	当使用在被导入模块中定义的函数时，必须包含模块的名字。所以不能只使用 `buildConnectionString`，而应该使用 `odbchelper.buildConnectionString`。如果您用过 Java 的类，对此应该不感到陌生。
[3]	访问函数的 `__doc__` 属性不像您想象的那样是通过函数调用。

> 注意
>
> 在 Python 中的 `import` 就像 Perl 中的 `require`。`import` 一个 Python 模块后，您就可以使用 `_module_._function_` 来访问它的函数；`require` 一个 Perl 模块后，您就可以使用 `_module_::_function_` 来访问它的函数。

2.4.1. 模块导入的搜索路径

在我们继续之前，我想简要地提一下库的搜索路径。当导入一个模块时，Python 在几个地方进行搜索。明确地，它会对定义在 `sys.path` 中的目录逐个进行搜索。它只是一个list (列表)，您可以容易地查看它或通过标准的list方法来修改它。(在本章的后面我们将学习更多关于

list的知识。)

例 2.4. 模块导入的搜索路径

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.2', '/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-dynload', '/usr/local/lib/python2.2/site-packages',
'/usr/local/lib/python2.2/site-packages/PIL', '/usr/local/lib/python2.2/site-packages/pid
>>> sys
<module 'sys' (built-in)>
>>> sys.path.append('/my/new/path')
```

[1]	导入 `sys` 模块，使得它的所有函数和属性都有效。
[2]	`sys.path` 是一个指定当前搜索路径的目录列表。(您的输出结果可能有所不同，这取决于您的操作系统、正在运行的 Python 版本和初始安装的位置。)Python 将搜索这些目录 (按顺序) 来查找一个与您正试着导入的模块名相匹配的 `.py` 文件。
[3]	实际上，我没说实话。真实情况要比这更复杂，因为不是所有的模块都保存为 `.py` 文件。有一些模块 (像 `sys`)，是“内置模块”，它们实际上是置于 Python 内部的。内置模块的行为如同一般的模块，但是它们的 Python 源代码是不可用的，因为它们不是用 Python 写的！(`sys` 模块是用 C 写的。)
[4]	在运行时，通过向 `sys.path` 追加目录名，就可以在 Python 的搜索路径中增加新的目录，然后当您导入模块时，Python 也会在那个目录中进行搜索。这个作用在 Python 运行时一直生效。(在 第 3 章 我们将讨论更多的关于 `append` 和其它的 list 方法。)

2.4.2. 何谓对象？

在 Python 中一切都是对象，并且几乎一切都有属性和方法。所有的函数都有一个内置的 `__doc__` 属性，它会返回在函数源代码中定义的 `doc string`；`sys` 模块是一个对象，它有一个叫作 `path` 的属性；等等。

我们仍然在回避问题的实质，究竟何谓对象？不同的编程语言以不同的方式定义 “对象” 。 某些语言中，它意味着所有 对象必须 有属性和方法；另一些语言中，它意味着所有的对象都可以子类化。在 Python 中，定义是松散的；某些对象既没有属性也没有方法 (关于这一点的说明在 第 3 章)，而且不是所有的对象都可以子类化 (关于这一点的说明在第 5 章)。但是万物皆对象从感性上可以解释为：一切都可以赋值给变量或作为参数传递给函数 (关于这一点的说明在第 4 章)。

这一点太重要了，所以我会在刚开始就不止一次地反复强调它，以免您没注意到：在 Python 中万物皆对象。字符串是对象。列表是对象。函数是对象。甚至模块也是对象，这一点我们很快会看到。

进一步阅读

- *Python Reference Manual* 确切解释了在 Python 中万物皆对象的含义，因为有些书生气十足的人，喜欢花时间讨论这类的问题。
- eff-bot 总结了 Python 对象.

2.5. 代码缩进

Python 函数没有明显的 `begin` 和 `end`，没有标明函数的开始和结束的花括号。唯一的分隔符是一个冒号 (`:`)，接着代码本身是缩进的。

例 2.5. 缩进 `buildConnectionString` 函数

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.
    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

代码块是通过它们的缩进来定义的。我所说的“代码块”是指：函数、 `if` 语句、 `for` 循环、 `while` 循环，等等。开始缩进表示块的开始，取消缩进表示块的结束。不存在明显的括号，大括号或关键字。这就意味着空白是重要的，并且要一致。在这个例子中，函数代码 (包括 `doc string`) 缩进了 4 个空格。不一定非要是 4 个，只要一致就可以了。没有缩进的第一行则被视为在函数体之外。

例 2.6 “if 语句” 展示了一个 `if` 语句缩进的例子。

例 2.6. `if` 语句

```
def fib(n):
    print 'n =', n
    if n > 1:
        return n * fib(n - 1)
    else:
        print 'end of the line'
        return 1
```

[1]	这是一个名为 `fib` 的函数，有一个参数 `n` 。在函数内的所有代码都是缩进的。
[2]	在 Python 中向屏幕输出内容非常容易，只要使用 `print` 即可。 `print` 语句可以接受任何数据类型，包括字符串、整数和其它类型，如字典和列表 (我们将在下一章学习)。甚至可以混在一起输出，只需用逗号隔开。所有值都输出到同一行，用空格隔开 (逗号并不打印出来)。所以当用 `5` 来调用 `fib` 时，将输出“n = 5”。
[3]	`if` 语句是一种的代码块。如果 `if` 表达式计算为 true，紧跟着的缩进块会被执行，否则进入 `else` 块执行。
[4]	当然 `if` 和 `else` 块可以包含许多行，只要它们都同样缩进。这个 `else` 块中有两行代码。对于多行代码块没有其它特殊的语法，只要缩进就行了。

在经过一些最初的抗议和几个与 Fortran 的嘲讽的类比之后，您会心平气和地对待代码缩进，并且开始看到它的好处。一个主要的好处就是所有的 Python 程序看上去都差不多，因为缩进是一种语言的要求而不是一种风格。这样就使得阅读和理解他人的 Python 代码容易得多。

> 注意
> Python 使用硬回车来分割语句，冒号和缩进来分割代码块。C++ 和 Java 使用分号来分割语句，花括号来分割代码块。

进一步阅读

- *Python Reference Manual* 讨论了交叉缩进问题，并且演示了各种各样的缩进错误。
- *Python Style Guide* 讨论了良好的缩进风格。

2.6. 测试模块

所有的 Python 模块都是对象，并且有几个有用的属性。您可以使用这些属性方便地测试您所编写的模块。下面是一个使用 `if` `__name__` 的技巧。

```
if __name__ == "__main__":
```

在继续学习新东西之前，有几个重要的观察结果。首先， `if` 表达式无需使用圆括号括起来。其次， `if` 语句以冒号结束， 紧跟其后的是缩进代码。

> 注意
> 与 C 一样，Python 使用 `==` 做比较，使用 `=` 做赋值。与 C 不一样，Python 不支持行内赋值，所以不会出现想要进行比较却意外地出现赋值的情况。

那么为什么说这个特殊的 `if` 语句是一个技巧呢？模块是对象，并且所有的模块都有一个内置属性 `__name__` 。一个模块的 `__name__` 的值取决于您如何应用模块。如果 `import` 模块，那么 `__name__` 的值通常为模块的文件名，不带路径或者文件扩展名。但是您也可以像一个标准的程序一样直接运行模块，在这种情况下 `__name__` 的值将是一个特别的缺省值， `__main__` 。

```
>>> import odbchelper
>>> odbchelper.`__name__`
'odbchelper'
```

只要了解到这一点，您就可以在模块内部为您的模块设计一个测试套件，在其中加入这个 `if` 语句。当您直接运行模块， `__name__` 的值是 `__main__` ，所以测试套件执行。当您导入模块， `__name__` 的值就是别的东西了，所以测试套件被忽略。这样使得在将新的模块集成到一个大程序之前开发和调试容易多了。

> 提示
> 在 MacPython 上，需要一个额外的步聚来使得 `if` `__name__` 技巧有效。点击窗口右上角的黑色三角，弹出模块的属性菜单，确认 Run as **main** 被选中。

进一步阅读

- *Python Reference Manual* 讨论了导入模块的底层细节。

第 3 章 内置数据类型

让我们用点儿时间来回顾一下您的第一个 Python 程序。但首先，先说些其他的内容，因为您需要了解一下 dictionary (字典)、tuple (元组) 和 list (列表)(哦，我的老天！)。如果您是一个 Perl hacker，当然可以撇开 dictionary 和 list，但是仍然需要注意 tuple。

3.1. Dictionary 介绍

Dictionary 是 Python 的内置数据类型之一，它定义了键和值之间一对一的关系。

> 注意
> Python 中的 dictionary 就像 Perl 中的 hash (哈希数组)。在 Perl 中，存储哈希值的变量总是以 `%` 字符开始；在 Python 中，变量可以任意取名，并且 Python 在内部会记录下其数据类型。
>
> 注意
> Python 中的 dictionary 像 Java 中的 `Hashtable` 类的实例。
>
> 注意
> Python 中的 dictionary 像 Visual Basic 中的 `Scripting.Dictionary` 对象的实例。

3.1.1. Dictionary 的定义

例 3.1. 定义 Dictionary

```
>>> d = {"server":"mpilgrim", "database":"master"}
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]
'mpilgrim'
>>> d["database"]
'master'
>>> d["mpilgrim"]
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

[1]	首先我们创建了新 dictionary，它有两个元素，将其赋给变量 `d` 。每一个元素都是一个 key-value 对；整个元素集合用大括号括起来。
[2]	`'server'` 是一个 key，它所关联的值是通过 `d["server"]` 来引用的，为 `'mpilgrim'` 。
[3]	`'database'` 是一个 key，它所关联的值是通过 `d["database"]` 来引用的，为 `'master'` 。
[4]	您可以通过 key 来引用其值，但是不能通过值获取 key。所以 `d["server"]` 的值为 `'mpilgrim'` ，而使用 `d["mpilgrim"]` 会引发一个异常，因为 `'mpilgrim'` 不是一个 key。

3.1.2. Dictionary 的修改

例 3.2. 修改 Dictionary

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs"
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

[1]	在一个 dictionary 中不能有重复的 key。给一个存在的 key 赋值会覆盖原有的值。
[2]	在任何时候都可以加入新的 key-value 对。这种语法同修改存在的值是一样的。(是的，它可能某天会给您带来麻烦。假设你一次次地修改一个 dictionary，但其间您使用的 key 并未按照您的想法进行改变。您可能以为加入了新值，但实际上只是一次又一次地修改了同一个值。)

请注意新的元素 (key 为 `'uid'` ，value 为 `'sa'`) 出现在中间。实际上，在第一个例子中的元素看上去是的有序不过是一种巧合。现在它们看上去的无序同样是一种巧合。

> 注意
> Dictionary 没有元素顺序的概念。说元素 “顺序乱了” 是不正确的，它们只是序偶的简单排列。这是一个重要的特性，它会在您想要以一种特定的，可重现的顺序 (像以 key 的字母表顺序) 存取 dictionary 元素的时候骚扰您。有一些实现这些要求的方法，它们只是没有加到 dictionary 中去。

当使用 dictionary 时，您需要知道：dictionary 的 key 是大小写敏感的。

例 3.3. Dictionary 的 key 是大小写敏感的

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "other value"
>>> d
{'key': 'other value'}
>>> d["Key"] = "third value"
>>> d
{'Key': 'third value', 'key': 'other value'}
```

[1]	为一个已经存在的 dictionary key 赋值，将简单覆盖原有的值。
[2]	这不会为一个已经存在的 dictionary key 赋值，因为在 Python 中是区分大小写的，也就是说 `'key'` 与 `'Key'` 是不同的。所以这种情况将在 dictionary 中创建一个新的 key-value 对。虽然看上去很相近，但是在 Python 眼里是完全不同的。

例 3.4. 在 dictionary 中混用数据类型

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
>>> d["retrycount"] = 3
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d[42] = "douglas"
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
42: 'douglas', 'retrycount': 3}
```

[1]	Dictionary 不只是用于存储字符串。Dictionary 的值可以是任意数据类型，包括字符串、整数、对象，甚至其它的 dictionary。在单个 dictionary 里，dictionary 的值并不需要全都是同一数据类型，可以根据需要混用和匹配。
[2]	Dictionary 的 key 要严格多了，但是它们可以是字符串、整数或几种其它的类型 (后面还会谈到这一点)。也可以在一个 dictionary 中混用和匹配 key 的数据类型。

3.1.3. 从 dictionary 中删除元素

例 3.5. 从 dictionary 中删除元素

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
42: 'douglas', 'retrycount': 3}
>>> del d[42]
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d.clear()
>>> d
{}
```

[1]	`del` 允许您使用 key 从一个 dictionary 中删除独立的元素。
[2]	`clear` 从一个 dictionary 中清除所有元素。注意空的大括号集合表示一个没有元素的 dictionary。

进一步阅读

- *How to Think Like a Computer Scientist* 讲授了 dictionary 和如何使用 dictionary 模拟稀疏矩阵。
- Python Knowledge Base 有许多使用 dictionary 的示例代码。
- Python Cookbook 讨论了如何通过 key 对 dictionary 的值进行排序。
- *Python Library Reference* 总结了所有的 dictionary 方法。

3.2. List 介绍

List 是 Python 中使用最频繁的数据类型。如果您对 list 仅有的经验就是在 Visual Basic 中的数组或 Powerbuilder 中的数据存储，那么就打起精神学习 Python 的 list 吧。

> 注意
> Python 的 list 如同 Perl 中的数组。在 Perl 中，用来保存数组的变量总是以 `@` 字符开始；在 Python 中，变量可以任意取名，并且 Python 在内部会记录下其数据类型。
>
> 注意
> Python 中的 list 更像 Java 中的数组 (您可以简单地这样理解，但 Python 中的 list 远比 Java 中的数组强大)。一个更好的类比是 `ArrayList` 类，它可以保存任意对象，并且可以在增加新元素时动态扩展。

3.2.1. List 的定义

例 3.6. 定义 List

```
>>> li = ["a", "b", "mpilgrim", "z", "example"]
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0]
'a'
>>> li[4]
'example'
```

[1]	首先我们定义了一个有 5 个元素的 list。注意它们保持着初始的顺序。这不是偶然。List 是一个用方括号包括起来的有序元素的集合。
[2]	List 可以作为以 0 下标开始的数组。任何一个非空 list 的第一个元素总是 `li[0]` 。
[3]	这个包含 5 个元素 list 的最后一个元素是 `li[4]`，因为列表总是从 0 开始。

例 3.7. 负的 list 索引

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[-1]
'example'
>>> li[-3]
'mpilgrim'
```

[1]	负数索引从 list 的尾部开始向前计数来存取元素。任何一个非空的 list 最后一个元素总是 `li[-1]` 。
[2]	如果负数索引使您感到糊涂，可以这样理解： `li[-n] == li[len(li) - n]` 。所以在这个 list 里， `li[-3] == li[5 - 3] == li[2]` 。

例 3.8. list 的分片 (slice)

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[1:3]
['b', 'mpilgrim']
>>> li[1:-1]
['b', 'mpilgrim', 'z']
>>> li[0:3]
['a', 'b', 'mpilgrim']
```

[1]	您可以通过指定 2 个索引得到 list 的子集，叫做一个 “slice” 。返回值是一个新的 list，它包含了 list 中按顺序从第一个 slice 索引 (这里为 `li[1]`) 开始，直到但是不包括第二个 slice 索引 (这里为 `li[3]`) 的所有元素。
[2]	如果一个或两个 slice 索引是负数，slice 也可以工作。如果对您有帮助，您可以这样理解：从左向右阅读 list，第一个 slice 索引指定了您想要的第一个元素，第二个 slice 索引指定了第一个您不想要的元素。返回的值为在其间的每个元素。
[3]	List 从 0 开始，所以 `li[0:3]` 返回 list 的前 3 个元素，从 `li[0]` 开始，直到但不包括 `li[3]` 。

例 3.9. Slice 简写

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3]
['a', 'b', 'mpilgrim']
>>> li[3:]
['z', 'example']
>>> li[:]
['a', 'b', 'mpilgrim', 'z', 'example']
```

[1]	如果左侧分片索引为 0，您可以将其省略，默认为 0。所以 `li[:3]` 同 例 3.8 “list 的分片 (slice)” 的 `li[0:3]` 是一样的。
[2]	同样的，如果右侧分片索引是 list 的长度，可以将其省略。所以 `li[3:]` 同 `li[3:5]` 是一样的，因为这个 list 有 5 个元素。
[3]	请注意这里的对称性。在这个包含 5 个元素的 list 中， `li[:3]` 返回前 3 个元素，而 `li[3:]` 返回后 2 个元素。实际上， `li[:n]` 总是返回前 `n` 个元素，而 `li[n:]` 将返回剩下的元素，不管 list 有多长。
[4]	如果将两个分片索引全部省略，这将包括 list 的所有元素。但是与原始的名为 `li` 的 list 不同，它是一个新 list，恰好拥有与 `li` 一样的全部元素。 `li[:]` 是生成一个 list 完全拷贝的一个简写。

3.2.2. 向 list 中增加元素

例 3.10. 向 list 中增加元素

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new")
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new")
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.extend(["two", "elements"])
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
```

[1]	`append` 向 list 的末尾追加单个元素。
[2]	`insert` 将单个元素插入到 list 中。数值参数是插入点的索引。请注意，list 中的元素不必唯一，现在有两个独立的元素具有 `'new'` 这个值， `li[2]` 和 `li[6]` 。
[3]	`extend` 用来连接 list。请注意不要使用多个参数来调用 `extend` ，要使用一个 list 参数进行调用。在本例中，这个 list 有两个元素。

例 3.11. `extend` (扩展) 与 `append` (追加) 的差别

```
>>> li = ['a', 'b', 'c']
>>> li.extend(['d', 'e', 'f'])
>>> li
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(li)
6
>>> li[-1]
'f'
>>> li = ['a', 'b', 'c']
>>> li.append(['d', 'e', 'f'])
>>> li
['a', 'b', 'c', ['d', 'e', 'f']]
>>> len(li)
4
>>> li[-1]
['d', 'e', 'f']
```

[1]	Lists 的两个方法 `extend` 和 `append` 看起来类似，但实际上完全不同。 `extend` 接受一个参数，这个参数总是一个 list，并且把这个 list 中的每个元素添加到原 list 中。
[2]	在这里 list 中有 3 个元素 (`'a'` 、 `'b'` 和 `'c'`)，并且使用另一个有 3 个元素 (`'d'` 、 `'e'` 和 `'f'`) 的 list 扩展之，因此新的 list 中有 6 个元素。
[3]	另一方面， `append` 接受一个参数，这个参数可以是任何数据类型，并且简单地追加到 list 的尾部。在这里使用一个含有 3 个元素的 list 参数调用 `append` 方法。
[4]	原来包含 3 个元素的 list 现在包含 4 个元素。为什么是 4 个元素呢？因为刚刚追加的最后一个元素本身是个 *list*。List 可以包含任何类型的数据，也包括其他的 list。这或许是您所要的结果，或许不是。如果您的意图是 `extend` ，请不要使用 `append` 。

3.2.3. 在 list 中搜索

例 3.12. 搜索 list

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.index("example")
5
>>> li.index("new")
2
>>> li.index("c")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li
False
```

[1]	`index` 在 list 中查找一个值的首次出现并返回索引值。
[2]	`index` 在 list 中查找一个值的首次 出现。这里 `'new'` 在 list 中出现了两次，在 `li[2]` 和 `li[6]` ，但 `index` 只返回第一个索引， `2` 。
[3]	如果在 list 中没有找到值，Python 会引发一个异常。这一点与大部分的语言截然不同，大部分语言会返回某个无效索引。尽管这种处理可能令人讨厌，但它仍然是件好事，因为它说明您的程序会由于源代码的问题而崩溃，好于在后面当您使用无效索引而引起崩溃。
[4]	要测试一个值是否在 list 内，使用 `in` 。如果值存在，它返回 `True` ，否则返为 `False` 。

> 注意
> 在 2.2.1 版本之前，Python 没有单独的布尔数据类型。为了弥补这个缺陷，Python 在布尔环境 (如 `if` 语句) 中几乎接受所有东西，遵循下面的规则：
>
> - `0` 为 false; 其它所有数值皆为 true。
> - 空串 (`""`) 为 false; 其它所有字符串皆为 true。
> - 空 list (`[]`) 为 false; 其它所有 list 皆为 true。
> - 空 tuple (`()`) 为 false; 其它所有 tuple 皆为 true。
> - 空 dictionary (`{}`) 为 false; 其它所有 dictionary 皆为 true。
>
> 这些规则仍然适用于 Python 2.2.1 及其后续版本，但现在您也可以使用真正的布尔值，它的值或者为 `True` 或者为 `False` 。请注意第一个字母是大写的；这些值如同在 Python 中的其它东西一样都是大小写敏感的。

3.2.4. 从 list 中删除元素

例 3.13. 从 list 中删除元素

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("z")
>>> li
['a', 'b', 'new', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("new")
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("c")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()
'elements'
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
```

[1]	`remove` 从 list 中删除一个值的首次出现。
[2]	`remove` 仅仅 删除一个值的首次出现。在这里， `'new'` 在 list 中出现了两次，但 `li.remove("new")` 只删除了 `'new'` 的首次出现。
[3]	如果在 list 中没有找到值，Python 会引发一个异常来响应 `index` 方法。
[4]	`pop` 是一个有趣的东西。它会做两件事：删除 list 的最后一个元素，然后返回删除元素的值。请注意，这与 `li[-1]` 不同，后者返回一个值但不改变 list 本身。也不同于 `li.remove(_value_)`，后者改变 list 但并不返回值。

3.2.5. 使用 list 的运算符

例 3.14. List 运算符

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new']
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two']
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3
>>> li
[1, 2, 1, 2, 1, 2]
```

[1]	Lists 也可以用 `+` 运算符连接起来。 `_list_ = _list_ + _otherlist_` 相当于 `_list_.extend(_otherlist_)`。但 `+` 运算符把一个新 (连接后) 的 list 作为值返回，而 `extend` 只修改存在的 list。也就是说，对于大型 list 来说， `extend` 的执行速度要快一些。
[2]	Python 支持 `+=` 运算符。 `li += ['two']` 等同于 `li.extend(['two'])`。 `+=` 运算符可用于 list、字符串和整数，并且它也可以被重载用于用户自定义的类中 (更多关于类的内容参见 第 5 章)。
[3]	`*` 运算符可以作为一个重复器作用于 list。 `li = [1, 2] * 3` 等同于 `li = [1, 2] + [1, 2] + [1, 2]`，即将三个 list 连接成一个。

进一步阅读

- *How to Think Like a Computer Scientist* 讲述了list，并且重点讲述了如何把 list 作为函数参数传递。
- *Python Tutorial* 展示了如何把 list 作为堆栈和队列使用。
- Python Knowledge Base 回答了有关 list 的常见问题并且有许多使用 list 的示例代码。
- *Python Library Reference* 总结了所有的 list 方法。

3.3. Tuple 介绍

Tuple 是不可变的 list。一旦创建了一个 tuple，就不能以任何方式改变它。

例 3.15. 定义 tuple

```
>>> t = ("a", "b", "mpilgrim", "z", "example")
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]
'a'
>>> t[-1]
'example'
>>> t[1:3]
('b', 'mpilgrim')
```

[1]	定义 tuple 与定义 list 的方式相同，但整个元素集是用小括号包围的，而不是方括号。
[2]	Tuple 的元素与 list 一样按定义的次序进行排序。Tuples 的索引与 list 一样从 0 开始，所以一个非空 tuple 的第一个元素总是 `t[0]` 。
[3]	负数索引与 list 一样从 tuple 的尾部开始计数。
[4]	与 list 一样分片 (slice) 也可以使用。注意当分割一个 list 时，会得到一个新的 list ；当分割一个 tuple 时，会得到一个新的 tuple。

例 3.16. Tuple 没有方法

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t
True
```

[1]	您不能向 tuple 增加元素。Tuple 没有 `append` 或 `extend` 方法。
[2]	您不能从 tuple 删除元素。Tuple 没有 `remove` 或 `pop` 方法。
[3]	您不能在 tuple 中查找元素。Tuple 没有 `index` 方法。
[4]	然而，您可以使用 `in` 来查看一个元素是否存在于 tuple 中。

那么使用 tuple 有什么好处呢？

- Tuple 比 list 操作速度快。如果您定义了一个值的常量集，并且唯一要用它做的是不断地遍历它，请使用 tuple 代替 list。
- 如果对不需要修改的数据进行 “写保护”，可以使代码更安全。使用 tuple 而不是 list 如同拥有一个隐含的 `assert` 语句，说明这一数据是常量。如果必须要改变这些值，则需要执行 tuple 到 list 的转换 (需要使用一个特殊的函数)。
- 还记得我说过 dictionary keys 可以是字符串，整数和 “其它几种类型”吗？Tuples 就是这些类型之一。Tuples 可以在 dictionary 中被用做 key，但是 list 不行。实际上，事情要比这更复杂。Dictionary key 必须是不可变的。Tuple 本身是不可改变的，但是如果您有一个 list 的 tuple，那就认为是可变的了，用做 dictionary key 就是不安全的。只有字符串、整数或其它对 dictionary 安全的 tuple 才可以用作 dictionary key。
- Tuples 可以用在字符串格式化中，我们会很快看到。

> 注意
> Tuple 可以转换成 list，反之亦然。内置的 `tuple` 函数接收一个 list，并返回一个有着相同元素的 tuple。而 `list` 函数接收一个 tuple 返回一个 list。从效果上看，`tuple` 冻结一个 list，而 `list` 解冻一个 tuple。

进一步阅读

- *How to Think Like a Computer Scientist* 讲解了 tuple 并且展示了如何连接 tuple。
- Python Knowledge Base 展示了如何对一个 tuple 排序。
- *Python Tutorial* 展示了如何定义一个只包含一个元素的 tuple。

3.4. 变量声明

现在您已经了解了有关 dictionary、tuple、和 list 的相关知识 (哦，我的老天！)，让我们回到 第 2 章 的例子程序 `odbchelper.py` 。

Python 与大多数其它语言一样有局部变量和全局变量之分，但是它没有明显的变量声明。变量通过首次赋值产生，当超出作用范围时自动消亡。

例 3.17. 定义 `myParams` 变量

```
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
```

首先注意缩进。 `if` 语句是代码块，需要像函数一样缩进。

其次，变量的赋值是一条被分成了多行的命令，用反斜线 (“ `\` ”) 作为续行符。

> 注意
> 当一条命令用续行符 (“ `\` ”) 分割成多行时，后续的行可以以任何方式缩进，此时 Python 通常的严格的缩进规则无需遵守。如果您的 Python IDE 自由对后续行进行了缩进，您应该把它当成是缺省处理，除非您有特别的原因不这么做。

严格地讲，在小括号，方括号或大括号中的表达式 (如定义一个 dictionary) 可以用或者不用续行符 (“ `\` ”) 分割成多行。甚至在不是必需的时候，我也喜欢使用续行符，因为我认为这样会让代码读起来更容易，但那只是风格问题。

第三，您从未声明过变量 `myParams` ，您只是给它赋了一个值。这点就像是 VBScript 没有设置 `option explicit` 选项一样。幸运的是，与 VBScript 不同，Python 不允许您引用一个未被赋值的变量，试图这样做会引发一个异常。

3.4.1. 变量引用

例 3.18. 引用未赋值的变量

```
>>> x
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'x'
>>> x = 1
>>> x
1
```

迟早有一天您会为此而感谢 Python 。

3.4.2. 一次赋多值

Python 中比较 “酷” 的一种编程简写是使用序列来一次给多个变量赋值。

例 3.19. 一次赋多值

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v
>>> x
'a'
>>> y
'b'
>>> z
'e'
```

[1]	`v` 是一个三元素的 tuple，并且 `(x, y, z)` 是一个三变量的 tuple。将一个 tuple 赋值给另一个 tuple，会按顺序将 `v` 的每个值赋值给每个变量。

这种用法有许多种用途。我经常想要将一定范围的值赋给多个变量。在 C 语言中，可以使用 `enum` 类型，手工列出每个常量和其所对应的值，当值是连续的时候这一过程让人感到特别繁琐。而在 Python 中，您可以使用内置的 `range` 函数和多变量赋值的方法来快速进行赋值。

例 3.20. 连续值赋值

```
>>> range(7)
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7)
>>> MONDAY
0
>>> TUESDAY
1
>>> SUNDAY
6
```

[1]	内置的 `range` 函数返回一个元素为整数的 list。这个函数的简化调用形式是接收一个上限值，然后返回一个初始值从 0 开始的 list，它依次递增，直到但不包含上限值。(如果您愿意，您可以传入其它的参数来指定一个非 `0` 的初始值和非 `1` 的步长。也可以使用 `print range.__doc__` 来了解更多的细节。)
[2]	`MONDAY`、`TUESDAY`、`WEDNESDAY`、`THURSDAY`、`FRIDAY`、`SATURDAY` 和 `SUNDAY` 是我们定义的变量。(这个例子来自 `calendar` 模块。它是一个很有趣的打印日历的小模块，像 UNIX 的 `cal` 命令。这个 `calendar` 模块定义了一星期中每天的整数常量表示。)
[3]	现在每个变量都拥有了自己的值：`MONDAY` 的值为 `0`，`TUESDAY` 的值为 `1`，等等。

您也可以使用多变量赋值来创建返回多个值的函数，只要返回一个包含所有值的 tuple 即可。调用者可以将其视为一个 tuple，或将值赋给独立的变量。许多标准的 Python 库都是这样做的，包括 `os` 模块，我们将在 第 6 章 中讨论。

进一步阅读

- *Python Reference Manual* 展示了什么时候可以忽略续行符和什么时候您需要使用续行符的例子。
- *How to Think Like a Computer Scientist* 演示了如何使用多变量赋值来交换两个变量的值。

3.5. 格式化字符串

Python 支持格式化字符串的输出 。尽管这样可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 `%s` 的字符串中。

> 注意
> 在 Python 中，字符串格式化使用与 C 中 `sprintf` 函数一样的语法。

例 3.21. 字符串的格式化

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v)
'uid=sa'
```

[1]	整个表达式的值为一个字符串。第一个 `%s` 被变量 `k` 的值替换；第二个 `%s` 被 `v` 的值替换。字符串中的所有其它字符 (在这个例子中，是等号) 按原样打印输出。

注意 `(k, v)` 是一个 tuple。我说过它们对某些东西有用。

您可能一直在想，做了这么多工作只不过是为了做简单的字符串连接。您想的不错，只不过字符串格式化不只是连接。它甚至不仅仅是格式化。它也是强制类型转换。

例 3.22. 字符串格式化与字符串连接的比较

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid)
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )
Users connected: 6
>>> print "Users connected: " + userCount
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

[1]	`+` 是字符串连接操作符。
[2]	在这个简单例子中，字符串格式化实现与连接一样的结果。
[3]	`(userCount, )` 是一个只包含一个元素的 tuple。是的，语法有一点奇怪，但是使用它的理由就是：显示地指出它是一个 tuple，而不是其他。实际上，当定义一个 list、tuple 或 dictionary 时，您可以总是在最后一个元素后面跟上一个逗号，但是当定义一个只包含一个元素的 tuple 时逗号是必须的。如果省略逗号，Python 不会知道 `(userCount)` 究竟是一个只包含一个元素的 tuple 还是变量 `userCount` 的值。
[4]	字符串格式化通过将 `%s` 替换成 `%d` 即可处理整数。
[5]	试图将一个字符串同一个非字符串连接会引发一个异常。与字符串格式化不同，字符串连接只能在被连接的每一个都是字符串时起作用。

如同 `printf` 在 C 中的作用，Python 中的字符串格式化是一把瑞士军刀。它有丰富的选项，不同的格式化格式符和可选的修正符可用于不同的数据类型。

例 3.23. 数值的格式化

```
>>> print "Today's stock price: %f" % 50.4625
50.462500
>>> print "Today's stock price: %.2f" % 50.4625
50.46
>>> print "Change since yesterday: %+.2f" % 1.5
+1.50
```

[1]	`%f` 格式符选项对应一个十进制浮点数，不指定精度时打印 6 位小数。
[2]	使用包含“.2”精度修正符的 `%f` 格式符选项将只打印 2 位小数。
[3]	您甚至可以混合使用各种修正符。添加 `+` 修正符用于在数值之前显示一个正号或负号。注意“.2”精度修正符仍旧在它原来的位置，用于只打印 2 位小数。

进一步阅读

- *Python Library Reference* 总结了所有字符串格式化所使用的格式符。
- _Effective AWK Programming_Top) 讨论了所有的格式符Control+Letters)和高级字符串格式化技术，如指定宽度，精度和0填充Format+Modifiers)。

3.6. 映射 list

Python 的强大特性之一是其对 list 的解析，它提供一种紧凑的方法，可以通过对 list 中的每个元素应用一个函数，从而将一个 list 映射为另一个 list。

例 3.24. List 解析介绍

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]
[2, 18, 16, 8]
>>> li
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li]
>>> li
[2, 18, 16, 8]
```

[1]	为了便于理解它，让我们从右向左看。 `li` 是一个将要映射的 list。Python 循环遍历 `li` 中的每个元素。对每个元素均执行如下操作：首先临时将其值赋给变量 `elem`， 然后 Python 应用函数 `` `elem `` *2` 进行计算，最后将计算结果追加到要返回的 list 中。
[2]	需要注意是， 对 list 的解析并不改变原始的 list。
[3]	将一个 list 的解析结果赋值给对其映射的变量是安全的。不用担心存在竞争情况或任何古怪事情的发生。Python 会在内存中创建新的 list，当对 list 的解析完成时，Python 将结果赋给变量。

让我们回过头来看看位于 第 2 章 的函数 `buildConnectionString` 对 list 的解析：

```
["%s=%s" % (k, v) for k, v in params.items()]
```

首先，注意到你调用了dictionary `params` 的 `items` 函数。这个函数返回一个 dictionary 中所有数据的 tuple 的 list。

例 3.25. `keys` , `values` 和 `items` 函数

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.keys()
['server', 'uid', 'database', 'pwd']
>>> params.values()
['mpilgrim', 'sa', 'master', 'secret']
>>> params.items()
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
```

[1]	Dictionary 的 `keys` 方法返回一个包含所有键的 list。这个 list 没按 dictionary 定义的顺序输出 (记住，元素在 dictionary 中是无序的)，但它是一个 list。
[2]	`values` 方法返回一个包含所有值的 list。它同 `keys` 方法返回的 list 输出顺序相同，所以对于所有的 `n` ， `params.values()[n] == params[params.keys()[n]]` 。
[3]	`items` 方法返回一个由形如 `(_key_, _value_)` 组成的 tuple 的 list。这个 list 包括 dictionary 中所有的数据。

现在让我们看一看 `buildConnectionString` 做了些什么。它接收一个 list， `params`.`items`()`，通过对每个元素应用字符串格式化将其映射为一个新 list。这个新 list 将与 params . items () 一一对应：新 list 中的每个元素都是 dictionary `params` 中的一个键-值对构成的的字符串。

例 3.26. `buildConnectionString` 中的 list 解析

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.items()
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
>>> [k for k, v in params.items()]
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()]
['mpilgrim', 'sa', 'master', 'secret']
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

[1]	请注意我们正在使用两个变量对 list `params.items()` 进行遍历。这是多变量赋值的另一种用法。 `params.items()` 的第一个元素是 `('server', 'mpilgrim')` ，所以在 list 解析的第一次遍历中， `k` 将为 `'server'` ， `v` 将为 `'mpilgrim'` 。在本例中，我们忽略了返回 list 中 `v` 的值，而只包含了 `k` 的值，所以这个 list 解析最后等于 `params . keys ()`。
[2]	这里我们做着相同的事情，但是忽略了 `k` 的值，所以这个 list 解析最后等于 `params . values ()`。
[3]	用一些简单的 字符串格式化将前面两个例子合并起来， 我们就得到一个包括了 dictionary 中每个元素的 key-value 对的 list。这个看上去有点像程序的输出结果，剩下的就只是将这个 list 中的元素接起来形成一个字符串了。

进一步阅读

- *Python Tutorial* 讨论了另一种方法来映射 list：使用内置的 `map` 函数。
- *Python Tutorial* 展示了如何对嵌套 list 的 list 进行解析。

3.7. 连接 list 与分割字符串

- 3.7.1. 字符串方法的历史注解

您有了一个形如 `_key_=_value_` 的 key-value 对 list，并且想将它们合成为单个字符串。为了将任意包含字符串的 list 连接成单个字符串，可以使用字符串对象的 `join` 方法。

下面是一个在 `buildConnectionString` 函数中连接 list 的例子：

```
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

在我们继续之前有一个有趣的地方。我一直在重复函数是对象，字符串是对象，每个东西都是对象的概念。您也许认为我的意思是说字符串值 是对象。但是不对，仔细地看一下这个例子，您将会看到字符串 `";"` 本身就是一个对象，您在调用它的 `join` 方法。

总之，这里的 `join` 方法将 list 中的元素连接成单个字符串，每个元素用一个分号隔开。分隔符不必是一个分号；它甚至不必是单个字符。它可以是任何字符串。

> 小心
> `join` 只能用于元素是字符串的 list；它不进行任何的强制类型转换。连接一个存在一个或多个非字符串元素的 list 将引发一个异常。

例 3.27. `odbchelper.py` 的输出结果

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> ";".join(["%s=%s" % (k, v) for k, v in params.items()])
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

上面的字符串是从 `odbchelper` 函数返回的，被调用块打印出来，这样就给出了您开始阅读本章时令人感到吃惊的输出结果。

您可能在想是否存在一个适当的方法来将字符串分割成一个 list。当然有，它叫做 `split` 。

例 3.28. 分割字符串

```
>>> li = ['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s = ";".join(li)
>>> s
'server=mpilgrim;uid=sa;database=master;pwd=secret'
>>> s.split(";")
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s.split(";", 1)
['server=mpilgrim', 'uid=sa;database=master;pwd=secret']
```

[1]	`split` 与 `join` 正好相反，它将一个字符串分割成多元素 list。注意，分隔符 (“ `;` ”) 被完全去掉了，它没有在返回的 list 中的任意元素中出现。
[2]	`split` 接受一个可选的第二个参数，它是要分割的次数。(“哦，可选参数……”，您将会在下一章中学会如何在您自己的函数中使用它。)

> 提示
>
> `_anystring_.`split`(_delimiter_, 1)` 是一个有用的技术，在您想要搜索一个子串，然后分别处理字符前半部分 (即 list 中第一个元素) 和后半部分 (即 list 中第二个元素) 时，使用这个技术。

进一步阅读

- Python Knowledge Base 回答了关于字符串的常见问题，并且有许多使用字符串的例子代码。
- *Python Library Reference* 总结了所有字符串方法。
- *Python Library Reference* 提供了 `string` 模块的文档。
- *The Whole Python FAQ* 解释了为什么 `join` 是字符串方法而不是 list 方法。

3.7.1. 字符串方法的历史注解

当我开始学 Python 时，我以为 `join` 是 list 的方法，它会使用分隔符作为一个参数。很多人都有同样的感觉：在 `join` 方法的背后有一段故事。在 Python 1.6 之前，字符串完全没有这些有用的方法。有一个独立的 `string` 模块包含所有的字符串函数，每个函数使用一个字符串作为它的第一个参数。这些函数被认为足够重要，所以它们移到字符串中去了，这就使得诸如 `lower` 、 `upper` 和 `split` 之类的函数是有意义的。但许多核心的 Python 程序员反对新的 `join` 方法，争论说应该换成是 list 的一个方法，或不应该移动而仅仅保留为旧的 `string` 模块 (现仍然还有许多有用的东西在里面) 的一部分。我只使用新的 `join` 方法，但是您还是会看到其它写法。如果它真的使您感到麻烦，您可以使用旧的 `string.join` 函数来替代。

3.8. 小结

现在 `odbchelper.py` 程序和它的输出结果都应该非常清楚了。

```
 def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.
    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

下面是 `odbchelper.py` 的输出结果：

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

在深入下一章学习之前，确保您可以无阻碍地完成下面的事情：

- 使用 Python IDE 来交互式地测试表达式
- 编写 Python 程序并且从 IDE 运行，或者从命令行运行
- 导入模块及调用它们的函数
- 声明函数以及 `doc string`、局部变量和适当的缩进的使用
- 定义 dictionary、tuple 和 list
- 任意一个对象的访问方法，包括：字符串、list、dictionary、函数和模块
- 通过字符串格式化连接值
- 使用 list 解析映射 list 为其他的 list
- 把字符串分割为 list 和把 list 连接为字符串

第 4 章 自省的威力

本章论述了 Python 众多强大功能之一：自省。正如你所知道的，Python 中万物皆对象，自省是指代码可以查看内存中以对象形式存在的其它模块和函数，获取它们的信息，并对它们进行操作。用这种方法，你可以定义没有名称的函数，不按函数声明的参数顺序调用函数，甚至引用事先并不知道名称的函数。

4.1. 概览

下面是一个完整可运行的 Python 程序。大概看一下这段程序，你应该可以理解不少了。用数字标出的行阐述了 第 2 章 第一个 *Python* 程序 中涉及的一些概念。如果剩下来的代码看起来有点奇怪，不用担心，通过阅读本章你将会理解所有这些。

例 4.1. `apihelper.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
def info(object, spacing=10, collapse=1):
    """Print methods and doc strings.
    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])
if __name__ == "__main__":
    print info.__doc__
```

[1]	该模块有一个声明为 `info` 的函数。根据它的函数声明可知，它有三个参数 ：`object` 、 `spacing` 和 `collapse` 。实际上后面两个参数都是可选参数，关于这点你很快就会看到。
[2]	`info` 函数有一个多行的 `doc string` ， 简要地描述了函数的功能。注意这里并没有提到返回值；单独使用这个函数只是为了这个函数产生的效果，并不是为了它的返回值。
[3]	函数内的代码是缩进形式的。
[4]	`if __name__` 技巧允许这个程序在自己独立运行时做些有用的事情，同时又不妨碍作为其它程序的模块使用。在这个例子中，程序只是简单地打印出 `info` 函数的 `doc string` 。
[5]	`if` 语句使用 `==` 进行比较，而且不需要括号。

`info` 函数的设计意图是提供给工作在 Python IDE 中的开发人员使用，它可以接受任何含有函数或者方法的对象 (比如模块，含有函数，又比如list，含有方法) 作为参数，并打印出对象的所有函数和它们的 `doc string` 。

例 4.2. `apihelper.py` 的用法示例

```
>>> from apihelper import info
>>> li = []
>>> info(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

缺省地，程序输出进行了格式化处理，以使其易于阅读。多行 `doc string` 被合并到单行中，要改变这个选项需要指定 *`collapse`* 参数的值为 `0`。如果函数名称长于10个字符，你可以将 *`spacing`* 参数的值指定为更大的值以使输出更容易阅读。

例 4.3. `apihelper.py` 的高级用法

```
>>> import odbchelper
>>> info(odbchelper)
buildConnectionString Build a connection string from a dictionary Returns string.
>>> info(odbchelper, 30)
buildConnectionString          Build a connection string from a dictionary Returns string
>>> info(odbchelper, 30, 0)
buildConnectionString          Build a connection string from a dictionary
    Returns string.
```

4.2. 使用可选参数和命名参数

Python 允许函数参数有缺省值；如果调用函数时不使用参数，参数将获得它的缺省值。此外，通过使用命名参数还可以以任意顺序指定参数。SQL Server Transact/SQL 中的存储过程也可以做到这些；如果你是脚本高手，你可以略过这部分。

`info` 函数就是这样一个例子，它有两个可选参数。

```
def info(object, spacing=10, collapse=1):
```

`spacing` 和 `collapse` 是可选参数，因为它们已经定义了缺省值。 `object` 是必备参数，因为它没有指定缺省值。如果调用 `info` 时只指定一个参数，那么 `spacing` 缺省为 `10` ，`collapse` 缺省为 `1` 。如果调用 `info` 时指定两个参数， `collapse` 依然默认为 `1` 。

假如你要指定 `collapse` 的值，但是又想要接受 `spacing` 的缺省值。在绝大部分语言中，你可能运气就不太好了，因为你需要使用三个参数来调用函数，这势必要重新指定 `spacing` 的值。但是在 Python 中，参数可以通过名称以任意顺序指定。

例 4.4. `info` 的有效调用

```
info(odbchelper)
info(odbchelper, 12)
info(odbchelper, collapse=0)
info(spacing=15, object=odbchelper)
```

[1]	只使用一个参数， `spacing` 使用缺省值 `10` ， `collapse` 使用缺省值 `1` 。
[2]	使用两个参数， `collapse` 使用缺省值 `1` 。
[3]	这里你显式命名了 `collapse` 并指定了它的值。 `spacing` 将依然使用它的缺省值 `10` 。
[4]	甚至必备参数 (例如 `object` ，没有指定缺省值) 也可以采用命名参数的方式，而且命名参数可以以任意顺序出现。

这些看上去非常累，除非你意识到参数不过是一个字典。“通常” 不使用参数名称的函数调用只是一个简写的形式，Python 按照函数声明中定义的的参数顺序将参数值和参数名称匹配起来。大部分时间，你会使用“通常”方式调用函数，但是如果你需要，总是可以提供附加的灵活性。

> 注意
>
> 调用函数时唯一必须做的事情就是为每一个必备参数指定值 (以某种方式)；以何种具体的方式和顺序都取决于你。

进一步阅读

- *Python Tutorial* 确切地讨论了何时、如何进行缺省参数赋值，这都和缺省值是一个 list 还是一个具有副作用的表达式有关。

4.3. 使用 `type`、`str`、`dir` 和其它内置函数

Python 有小部分相当有用的内置函数。除这些函数之外，其它所有的函数都被分到了各个模块中。其实这是一个非常明智的设计策略，避免了核心语言变得像其它脚本语言一样臃肿 (咳咳，Visual Basic)。

4.3.1. `type` 函数

`type` 函数返回任意对象的数据类型。在 `types` 模块中列出了可能的数据类型。这对于处理多种数据类型的帮助者函数 [1] 非常有用。

例 4.5. `type` 介绍

```
>>> type(1)
<type 'int'>
>>> li = []
>>> type(li)
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)
<type 'module'>
>>> import types
>>> type(odbchelper) == types.ModuleType
True
```

[1]	`type` 可以接收任何东西作为参数——我的意思是任何东西——并返回它的数据类型。整型、字符串、列表、字典、元组、函数、类、模块，甚至类型对象都可以作为参数被 `type` 函数接受。
[2]	`type` 可以接收变量作为参数，并返回它的数据类型。
[3]	`type` 还可以作用于模块。
[4]	你可以使用 `types` 模块中的常量来进行对象类型的比较。这就是 `info` 函数所做的，很快你就会看到。

4.3.2. `str` 函数

`str` 将数据强制转换为字符串。每种数据类型都可以强制转换为字符串。

例 4.6. `str` 介绍

```
>>> str(1)
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen
['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen)
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper)
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)
'None'
```

[1]	对于简单的数据类型比如整型，你可以预料到 `str` 的正常工作，因为几乎每种语言都有一个将整型转化为字符串的函数。
[2]	然而 `str` 可以作用于任何数据类型的任何对象。这里它作用于一个零碎构建的列表。
[3]	`str` 还允许作用于模块。注意模块的字符串形式表示包含了模块在磁盘上的路径名，所以你的显示结果将会有所不同。
[4]	`str` 的一个细小但重要的行为是它可以作用于 `None` ， `None` 是 Python 的 null 值。这个调用返回字符串 `'None'` 。你将会使用这一点来改进你的 `info` 函数，这一点你很快就会看到。

`info` 函数的核心是强大的 `dir` 函数。 `dir` 函数返回任意对象的属性和方法列表，包括模块对象、函数对象、字符串对象、列表对象、字典对象 …… 相当多的东西。

例 4.7. `dir` 介绍

```
>>> li = []
>>> dir(li)
['append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d)
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault', 'update', 'values']
>>> import odbchelper
>>> dir(odbchelper)
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```

[1]	`li` 是一个列表，所以 `` `dir `` (`li`)` 返回一个包含所有列表方法的列表。注意返回的
[2]	`d` 是一个字典，所以 `` `dir `` (`d`) 返回字典方法的名称列表。其中至少有一个方法， `[` keys`](../na
[3]	这里就是真正变得有趣的地方。 `odbchelper` 是一个模块，所以 `` `dir `` (`odbchelper`) 返回模块中定义的所有部件的列表，包括内置的属性，例如 `[` **name** `](../getti` 函数。

最后是 `callable` 函数，它接收任何对象作为参数，如果参数对象是可调用的，返回 `True` ；否则返回 `False` 。可调用对象包括函数、类方法，甚至类自身 (下一章将更多的关注类)。

例 4.8. `callable` 介绍

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join
<function join at 00C55A7C>
>>> callable(string.punctuation)
False
>>> callable(string.join)
True
>>> print string.join.__doc__
join(list [,sep]) -> string
    Return a string composed of the words in list, with
    intervening occurrences of sep.  The default separator is a
    single space.
    (joinfields and join are synonymous)
```

[1]	`string` 模块中的函数现在已经不赞成使用了 (尽管很多人现在仍然还在使用 `join` 函数)，但是在这个模块中包含了许多有用的变量，例如 `string.punctuation` ，这个字符串包含了所有标准的标点符号字符。
[2]	`string.join` 是一个用于连接字符串列表的函数。
[3]	`string.punctuation` 是不可调用的对象；它是一个字符串。(字符串确有可调用的方法，但是字符串本身不是可调用的。)
[4]	`string.join` 是可调用的；这个函数可以接受两个参数。
[5]	任何可调用的对象都有 `doc string` 。通过将 `callable` 函数作用于一个对象的每个属性，可以确定哪些属性 (方法、函数、类) 是你要关注的，哪些属性 (常量等等) 是你可以忽略、之前不需要知道的。

4.3.3. 内置函数

`type` 、 `str` 、 `dir` 和其它的 Python 内置函数都归组到了 `__builtin__` (前后分别是双下划线) 这个特殊的模块中。如果有帮助的话，你可以认为 Python 在启动时自动执行了 `from __builtin__ import *` ，此语句将所有的 “内置” 函数导入该命名空间，所以在这个命名空间中可以直接使用这些内置函数。

像这样考虑的好处是，你是可以获取 `__builtin__` 模块信息的，并以组的形式访问所有的内置函数和属性。猜到什么了吗，现在我们的 Python 有一个称为 `info` 的函数。自己尝试一下，略看一下结果列表。后面我们将深入到一些更重要的函数。(一些内置的错误类，比如 `AttributeError` ，应该看上去已经很熟悉了。)

例 4.9. 内置属性和内置函数

```
>>> from apihelper import info
>>> import __builtin__
>>> info(__builtin__, 20)
ArithmeticError      Base class for arithmetic errors.
AssertionError       Assertion failed.
AttributeError       Attribute not found.
EOFError             Read beyond end of file.
EnvironmentError     Base class for I/O related errors.
Exception            Common base class for all exceptions.
FloatingPointError   Floating point operation failed.
IOError              I/O operation failed.
[...snip...]
```

> 注意
> Python 提供了很多出色的参考手册，你应该好好地精读一下所有 Python 提供的必备模块。对于其它大部分语言，你会发现自己要常常回头参考手册或者 man 页来提醒自己如何使用这些模块，但是 Python 不同于此，它很大程度上是自文档化的。

进一步阅读

- *Python Library Reference* 对所有的内置函数和所有的内置异常都进行了文档化。

Footnotes

[1] 帮助者函数，原文是 helper function，也就是我们在前文所看到的诸如 `odbchelper`、`apihelper` 这样的函数。——译注

4.4. 通过 `getattr` 获取对象引用

- 4.4.1. 用于模块的 getattr
- 4.4.2. getattr 作为一个分发者

你已经知道 Python 函数是对象。你不知道的是，使用 `getattr` 函数，可以得到一个直到运行时才知道名称的函数的引用。

例 4.10. `getattr` 介绍

```
>>> li = ["Larry", "Curly"]
>>> li.pop
<built-in method pop of list object at 010DF884>
>>> getattr(li, "pop")
<built-in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe")
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")
<built-in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

[1]	该语句获取列表的 `pop` 方法的引用。注意该语句并不是调用 `pop` 方法；调用 `pop` 方法的应该是 `li.pop()` 。这里指的是方法对象本身。
[2]	该语句也是返回 `pop` 方法的引用，但是此时，方法名称是作为一个字符串参数传递给 `getattr` 函数的。 `getattr` 是一个有用到令人无法致信的内置函数，可以返回任何对象的任何属性。在这个例子中，对象是一个 list，属性是 `pop` 方法。
[3]	如果不确信它是多么的有用，试试这个：`getattr` 的返回值是 方法，然后你就可以调用它，就像直接使用 `li.append("Moe")` 一样。但是实际上你没有直接调用函数；只是以字符串形式指定了函数名称。
[4]	`getattr` 也可以作用于字典。
[5]	理论上，`getattr` 可以作用于元组，但是由于元组没有方法，所以不管你指定什么属性名称 `getattr` 都会引发一个异常。

4.4.1. 用于模块的 `getattr`

`getattr` 不仅仅适用于内置数据类型，也可作用于模块。

例 4.11. `apihelper.py` 中的 `getattr` 函数

```
>>> import odbchelper
>>> odbchelper.buildConnectionString
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString")
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method)
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method))
<type 'function'>
>>> import types
>>> type(getattr(object, method)) == types.FunctionType
True
>>> callable(getattr(object, method))
True
```

[1]	该语句返回 `odbchelper` 模块中 `buildConnectionString` 函数的引用，第 2 章 第一个 *Python* 程序 你已经研习过这个方法了。(你看到的这个十六进制地址是我机器上的；你的输出结果会有所不同。)
[2]	使用 `getattr` ，你能够获得同一函数的同一引用。通常， `getattr (*object*, "*attribute*") 等价于 *object.attribute* 。如果 _ object _ 是一个模块的话，那么 _ attribute`_ 可能是定义在模块中的任何东西：函数、 类或者全局变量。
[3]	接下来的是你真正用在 `info` 函数中的东西。 `object` 作为一个参数传递给函数； `method` 是方法或者函数的名称字符串。
[4]	在这个例子中， `method` 是函数的名称，通过获取 `type` 可以进行验证。
[5]	由于 `method` 是一个函数，所以它是可调用的。

4.4.2. `getattr` 作为一个分发者

`getattr` 常见的使用模式是作为一个分发者。举个例子，如果你有一个程序可以以不同的格式输出数据，你可以为每种输出格式定义各自的格式输出函数，然后使用唯一的分发函数调用所需的格式输出函数。

例如，让我们假设有一个以 HTML、XML 和普通文本格式打印站点统计的程序。输出格式在命令行中指定，或者保存在配置文件中。 `statsout` 模块定义了三个函数： `output_html` 、 `output_xml` 和 `output_text` 。然后主程序定义了唯一的输出函数，如下：

例 4.12. 使用 `getattr` 创建分发者

```
 import statsout
def output(data, format="text"):
    output_function = getattr(statsout, "output_%s" % format)
    return output_function(data)
```

[1]	`output` 函数接收一个必备参数 `data`，和一个可选参数 `format`。如果没有指定 `format` 参数，其缺省值是 `text` 并完成普通文本输出函数的调用。
[2]	你可以连接 `format` 参数值和 "output_" 来创建一个函数名称作为参数值，然后从 `statsout` 模块中取得该函数。这种方式允许今后很容易地扩展程序以支持其它的输出格式，而且无需修改分发函数。所要做的仅仅是向 `statsout` 中添加一个函数，比如 `output_pdf`，之后只要将 “pdf” 作为 `format` 的参数值传递给 `output` 函数即可。
[3]	现在你可以简单地调用输出函数，就像调用其它函数一样。`output_function` 变量是指向 `statsout` 模块中相应函数的引用。

你是否发现前面示例的一个 Bug？即字符串和函数之间的松耦合，而且没有错误检查。如果用户传入一个格式参数，但是在 `statsout` 中没有定义相应的格式输出函数，会发生什么呢？还好，`getattr` 会返回 `None`，它会取代一个有效函数并被赋值给 `output_function`，然后下一行调用函数的语句将会失败并抛出一个异常。这种方式不好。

值得庆幸的是，`getattr` 能够使用可选的第三个参数，一个缺省返回值。

例 4.13. `getattr` 缺省值

```
 import statsout
def output(data, format="text"):
    output_function = getattr(statsout, "output_%s" % format, statsout.output_text)
    return output_function(data)
```

[1]	这个函数调用一定可以工作，因为你在调用 `getattr` 时添加了第三个参数。第三个参数是一个缺省返回值，如果第二个参数指定的属性或者方法没能找到，则将返回这个缺省返回值。

正如你所看到，`getattr` 是相当强大的。它是自省的核心，在后面的章节中你将看到它更强大的示例。

4.5. 过滤列表

如你所知，Python 具有通过列表解析 (第 3.6 节 “映射 list”) 将列表映射到其它列表的强大能力。这种能力同过滤机制结合使用，使列表中的有些元素被映射的同时跳过另外一些元素。

过滤列表语法：

```
[`mapping-expression` for `element` in `source-list` if `filter-expression`]
```

这是你所知所爱的列表解析的扩展。前三部分都是相同的；最后一部分，以 `if` 开头的是过滤器表达式。过滤器表达式可以是返回值为真或者假的任何表达式 (在 Python 中是几乎任何东西)。任何经过滤器表达式演算值为真的元素都可以包含在映射中。其它的元素都将忽略，它们不会进入映射表达式，更不会包含在输出列表中。

例 4.14. 列表过滤介绍

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"]
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
>>> [elem for elem in li if li.count(elem) == 1]
['a', 'mpilgrim', 'foo', 'c']
```

[1]	这里的映射表达式很简单 (只是返回每个元素的值)，所以请把注意力集中到过滤器表达式上。由于 Python 会遍历整个列表，它将对每个元素执行过滤器表达式。如果过滤器表达式演算值为真，该元素就会被映射，同时映射表达式的结果将包含在返回的列表中。这里，你过滤掉了所有单字符的字符串，留下了一个由长字符串构成的列表。
[2]	这里你过滤掉了一个特定值 `b` 。注意这个过滤器会过滤掉所有的 `b` ，因为每次取出 `b` ，过滤表达式都将为假。
[3]	`count` 是一个列表方法，返回某个值在列表中出现的次数。你可以认为这个过滤器将从列表中剔除重复元素，返回一个只包含了在原始列表中有着唯一值拷贝的列表。但并非如此，因为在原始列表中出现两次的值 (在本例中， `b` 和 `d`) 被完全剔除了。从一个列表中排除重复值有多种方法，但过滤并不是其中的一种。

回到 `apihelper.py` 中的这一行：

```
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
```

这行看上去挺复杂——确实也很复杂——但是基本结构都还是一样的。整个过滤表达式返回一个列表，并赋值给 `methodList` 变量。表达式的前半部分是列表映射部分。映射表达式是一个和遍历元素相同的表达式，因此它返回每个元素的值。 `` `dir `` (`object`) 返回 object 对象的属性和方法列表—你正在映射的列表。所以唯一新出现的部分就是在 if` 后面的过滤表达式。

过滤表达式看上去很恐怖，其实不是。你已经知道了 `callable` 、 `getattr` 和 `in` 。正如你在前面的部分中看到的，如果 `object` 是一个模块，并且 `method` 是上述模块中某个函数的名称，那么表达式 `getattr(object, method)` 将返回一个函数对象。

所以这个表达式接收一个名为 `object` 的对象，然后得到它的属性、方法、函数和其他成员的名称列表，接着过滤掉我们不关心的成员。执行过滤行为是通过对每个属性/方法/函数的名称调用 `getattr` 函数取得实际成员的引用，然后检查这些成员对象是否是可调用的，当然这些可调用的成员对象可能是方法或者函数，同时也可能是内置的 (比如列表的 `pop` 方法) 或者用户自定义的 (比如 `odbchelper` 模块的 `buildConnectionString` 函数)。这里你不用关心其它的属性，如内置在每一个模块中的 `__name__` 属性。

进一步阅读

- *Python Tutorial* 讨论了使用内置 `filter` 函数过滤列表的另一种方式。

4.6. `and` 和 `or` 的特殊性质

- 4.6.1. 使用 and-or 技巧

在Python 中， `and` 和 `or` 执行布尔逻辑演算，如你所期待的一样。但是它们并不返回布尔值，而是返回它们实际进行比较的值之一。

例 4.15. `and` 介绍

```
>>> 'a' and 'b'
'b'
>>> '' and 'b'
''
>>> 'a' and 'b' and 'c'
'c'
```

[1]	使用 `and` 时，在布尔环境中从左到右演算表达式的值。 `0` 、 `''` 、 `[]` 、 `()` 、 `{}` 、 `None` 在布尔环境中为假；其它任何东西都为真。还好，几乎是所有东西。默认情况下，布尔环境中的类实例为真，但是你可以在类中定义特定的方法使得类实例的演算值为假。你将会在第 5 章中了解到类和这些特殊方法。如果布尔环境中的所有值都为真，那么 `and` 返回最后一个值。在这个例子中， `and` 演算 `'a'` 的值为真，然后是 `'b'` 的演算值为真，最终返回 `'b'` 。
[2]	如果布尔环境中的某个值为假，则 `and` 返回第一个假值。在这个例子中， `''` 是第一个假值。
[3]	所有值都为真，所以 `and` 返回最后一个真值， `'c'` 。

例 4.16. `or` 介绍

```
>>> 'a' or 'b'
'a'
>>> '' or 'b'
'b'
>>> '' or [] or {}
{}
>>> def sidefx():
... print "in sidefx()"
... return 1
>>> 'a' or sidefx()
'a'
```

[1]	使用 `or` 时，在布尔环境中从左到右演算值，就像 `and` 一样。如果有一个值为真， `or` 立刻返回该值。本例中， `'a'` 是第一个真值。
[2]	`or` 演算 `''` 的值为假，然后演算 `'b'` 的值为真，于是返回 `'b'` 。
[3]	如果所有的值都为假， `or` 返回最后一个假值。 `or` 演算 `''` 的值为假，然后演算 `[]` 的值为假，依次演算 `{}` 的值为假，最终返回 `{}` 。
[4]	注意 `or` 在布尔环境中会一直进行表达式演算直到找到第一个真值，然后就会忽略剩余的比较值。如果某些值具有副作用，这种特性就非常重要了。在这里，函数 `sidefx` 永远都不会被调用，因为 `or` 演算 `'a'` 的值为真，所以紧接着就立刻返回 `'a'` 了。

如果你是一名 C 语言黑客，肯定很熟悉 `_bool_ ?` a `:` b ` 表达式，如果 _ bool _ 为真，表达式演算值为 a ，否则为 b 。基于 Python 中 and 和 or` 的工作方式，你可以完成相同的事情。

4.6.1. 使用 `and-or` 技巧

例 4.17. `and-or` 技巧介绍

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b
'first'
>>> 0 and a or b
'second'
```

[1]	这个语法看起来类似于 C 语言中的 `_bool_ ?` a `:` b ` 表达式。整个表达式从左到右进行演算，所以先进行 and 表达式的演算。 1 and 'first' 演算值为 'first' ，然后 'first' or 'second' 的演算值为 'first'`。
[2]	`0 and 'first'` 演算值为 `False` ，然后 `0 or 'second'` 演算值为 `'second'` 。

然而，由于这种 Python 表达式单单只是进行布尔逻辑运算，并不是语言的特定构成，这是 `and-or` 技巧和 C 语言中的 `_bool_ ?` a `:` b ` 语法非常重要的不同。如果 a` 为假，表达式就不会按你期望的那样工作了。(你能知道我被这个问题折腾过吗？不止一次？)

例 4.18. `and-or` 技巧无效的场合

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b
'second'
```

[1]	由于 `a` 是一个空字符串，在 Python 的布尔环境中空字符串被认为是假的，`1 and ''` 的演算值为 `''`，最后 `'' or 'second'` 的演算值为 `'second'`。噢！这个值并不是你想要的。

`and-or` 技巧，也就是

`_bool_ and a or b 表达式，当 `a` 在布尔环境中的值为假时，不会像 C 语言表达式 `_bool_ ? `a` : `b` 那样工作。

在 `and-or` 技巧后面真正的技巧是，确保 `a` 的值决不会为假。最常用的方式是使 `a` 成为 `[ a ]` 、 `b` 成为 `[ b ]`，然后使用返回值列表的第一个元素，应该是 `a` 或 `b` 中的某一个。

例 4.19. 安全使用 `and-or` 技巧

```
>>> a = ""
>>> b = "second"
>>> (1 and [a] or [b])[0]
''
```

[1]	由于 `[ a ]` 是一个非空列表，所以它决不会为假。即使 `a` 是 `0` 或者 `''` 或者其它假值，列表 `[ a ]` 也为真，因为它有一个元素。

到现在为止，这个技巧可能看上去问题超过了它的价值。毕竟，使用 `if` 语句可以完成相同的事情，那为什么要经历这些麻烦事呢？哦，在很多情况下，你要在两个常量值中进行选择，由于你知道 `a` 的值总是为真，所以你可以使用这种较为简单的语法而且不用担心。对于使用更为复杂的安全形式，依然有很好的理由要求这样做。例如，在 Python 语言的某些情况下 `if` 语句是不允许使用的，比如在 `lambda` 函数中。

进一步阅读

- Python Cookbook 讨论了其它的 `and-or` 技巧。

4.7. 使用 `lambda` 函数

- 4.7.1. 真实世界中的 lambda 函数

Python 支持一种有趣的语法，它允许你快速定义单行的最小函数。这些叫做 `lambda` 的函数，是从 Lisp 借用来的，可以用在任何需要函数的地方。

例 4.20. `lambda` 函数介绍

```
>>> def f(x):
... return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2
>>> g(3)
6
>>> (lambda x: x*2)(3)
6
```

[1]	这是一个 `lambda` 函数，完成同上面普通函数相同的事情。注意这里的简短的语法：在参数列表周围没有括号，而且忽略了 `return` 关键字 (隐含存在，因为整个函数只有一行)。而且，该函数没有函数名称，但是可以将它赋值给一个变量进行调用。
[2]	使用 `lambda` 函数时甚至不需要将它赋值给一个变量。这可能不是世上最有用的东西，它只是展示了 `lambda` 函数只是一个内联函数。

总的来说，`lambda` 函数可以接收任意多个参数 (包括可选参数) 并且返回单个表达式的值。`lambda` 函数不能包含命令，包含的表达式不能超过一个。不要试图向 `lambda` 函数中塞入太多的东西；如果你需要更复杂的东西，应该定义一个普通函数，然后想让它多长就多长。

> 注意
>
> `lambda` 函数是一种风格问题。不一定非要使用它们；任何能够使用它们的地方，都可以定义一个单独的普通函数来进行替换。我将它们用在需要封装特殊的、非重用代码上，避免令我的代码充斥着大量单行函数。

4.7.1. 真实世界中的 `lambda` 函数

`apihelper.py` 中的 `lambda` 函数：

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

注意这里使用了 `and-or` 技巧的简单形式，它是没问题的，因为 `lambda` 函数在布尔环境中总是为真。(这并不意味这 `lambda` 函数不能返回假值。这个函数对象的布尔值为真；它的返回值可以是任何东西。)

还要注意的是使用了没有参数的 `split` 函数。你已经看到过它带一个或者两个参数的使用，但是不带参数它按空白进行分割。

例 4.21. `split` 不带参数

```
>>> s = "this   is\na\ttest"
>>> print s
this   is
a    test
>>> print s.split()
['this', 'is', 'a', 'test']
>>> print " ".join(s.split())
'this is a test'
```

[1]	这是一个多行字符串，通过使用转义字符的定义代替了三重引号。 `\n` 是一个回车， `\t` 是一个制表符。
[2]	不带参数的 `split` 按照空白进行分割。所以三个空格、一个回车和一个制表符都是一样的。
[3]	通过 `split` 分割字符串你可以将空格统一化；然后再以单个空格作为分隔符用 `join` 将其重新连接起来。这也就是 `info` 函数将多行 `doc string` 合并成单行所做的事情。

那么 `info` 函数到底用这些 `lambda` 函数、 `split` 函数和 `and-or` 技巧做了些什么呢？

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

`processFunc` 现在是一个函数，但是它到底是哪一个函数还要取决于 `collapse` 变量。如果 `collapse` 为真， `processFunc`(_string_)` 将压缩空白；否则 processFunc `(_string_)` 将返回未改变的参数。

在一个不很健壮的语言中实现它，像 Visual Basic，你很有可能要创建一个函数，接受一个字符串参数和一个 *`collapse`* 参数，并使用 `if` 语句确定是否压缩空白，然后再返回相应的值。这种方式是低效的，因为函数可能需要处理每一种可能的情况。每次你调用它，它将不得不在给出你所想要的东西之前，判断是否要压缩空白。在 Python 中，你可以将决策逻辑拿到函数外面，而定义一个裁减过的 `lambda` 函数提供确切的 (唯一的) 你想要的。这种方式更为高效、更为优雅，而且很少引起那些令人讨厌 (哦，想到那些参数就头昏) 的错误。

`lambda` 函数进一步阅读

- Python Knowledge Base 讨论了使用 `lambda` 来间接调用函数。
- *Python Tutorial* 演示了如何从一个 `lambda` 函数内部访问外部变量。(PEP 227 解释了在 Python 的未来版本中将如何变化。)
- *The Whole Python FAQ* 有关于令人模糊的使用 `lambda` 单行语句的例子。

4.8. 全部放在一起

最后一行代码是唯一还没有解释过的，它完成全部的工作。但是现在工作已经简单了，因为所需要的每件事都已经按照需求建立好了。所有的多米诺骨牌已经就位，到了将它们推倒的时候了。

下面是 `apihelper.py` 的关键

```
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])
```

注意这是一条命令，被分隔成了多行，但是并没有使用续行符 (`\`)。还记得我说过一些表达式可以分割成多行而不需要使用反斜线吗？列表解析就是这些表达式之一，因为整个表达式包括在方括号里。

现在，让我们从后向前分析。

```
for method in methodList
```

告诉我们这是一个列表解析。如你所知 `methodList` 是 `object` 中所有你关心的方法的一个列表。所以你正在使用 `method` 遍历列表。

例 4.22. 动态得到 `doc string`

```
>>> import odbchelper
>>> object = odbchelper
>>> method = 'buildConnectionString'
>>> getattr(object, method)
<function buildConnectionString at 010D6D74>
>>> print getattr(object, method).__doc__
Build a connection string from a dictionary of parameters.

    Returns string.
```

[1]	在 `info` 函数中， `object` 是要得到帮助的对象，作为一个参数传入。
[2]	在你遍历 `methodList` 时， `method` 是当前方法的名称。
[3]	通过 `getattr` 函数，你可以得到 *`object`* 模块中 *`method`* 函数的引用。
[4]	现在，很容易就可以打印出方法的 `doc string` 。

接下来令人困惑的是 `doc string` 周围 `str` 的使用。你可能记得，`str` 是一个内置函数，它可以强制将数据转化为字符串。但是一个 `doc string` 应该总是一个字符串，为什么还要费事地使用 `str` 函数呢？答案就是：不是每个函数都有 `doc string` ，如果没有，这个 `__doc__` 属性为 `None` 。

例 4.23. 为什么对一个 `doc string` 使用 `str` ？

```
>>> >>> def foo(): print 2
>>> >>> foo()
2
>>> >>> foo.__doc__
>>> foo.__doc__ == None
True
>>> str(foo.__doc__)
'None'
```

[1]	你可以很容易的定义一个没有 `doc string` 的函数，这种情况下它的 `__doc__` 属性为 `None` 。令人迷惑的是，如果你直接演算 `__doc__` 属性的值，Python IDE 什么都不会打印。这是有意义的 (前提是你考虑了这个结果的来由)，但是却没有什么用。
[2]	你可以直接通过 `__doc__` 属性和 `None` 的比较验证 `__doc__` 属性的值。
[3]	`str` 函数可以接收值为 null 的参数，然后返回它的字符串表示，`'None'` 。

> 注意
> 在 SQL 中，你必须使用 `IS NULL` 代替 `= NULL` 进行 null 值比较。在 Python，你可以使用 `== None` 或者 `is None` 进行比较，但是 `is None` 更快。

现在你确保有了一个字符串，可以把这个字符串传给 `processFunc` ，这个函数已经定义是一个既可以压缩空白也可以不压缩空白的函数。现在你看出来为什么使用 `str` 将 `None` 转化为一个字符串很重要了。`processFunc` 假设接收到一个字符串参数然后调用 `split` 方法，如果你传入 `None` ，将导致程序崩溃，因为 `None` 没有 `split` 方法。

再往回走一步，你再一次使用了字符串格式化来连接 `processFunc` 的返回值 和 `method` 的 `ljust` 方法的返回值。`ljust` 是一个你之前没有见过的新字符串方法。

例 4.24. `ljust` 方法介绍

```
>>> s = 'buildConnectionString'
>>> s.ljust(30)
'buildConnectionString         '
>>> s.ljust(20)
'buildConnectionString'
```

[1]	`ljust` 用空格填充字符串以符合指定的长度。 `info` 函数使用它生成了两列输出并将所有在第二列的 `doc string` 纵向对齐。
[2]	如果指定的长度小于字符串的长度， `ljust` 将简单地返回未变化的字符串。它决不会截断字符串。

几乎已经完成了。有了 `ljust` 方法填充过的方法名称和来自调用 `processFunc` 方法得到的 `doc string` (可能压缩过)，你就可以将两者连接起来并得到单个字符串。因为对 `methodList` 进行了映射，最终你将获得一个字符串列表。利用 `"\n"` 的 `join` 函数，将这个列表连接为单个字符串，列表中每个元素独占一行，接着打印出结果。

例 **4.25.** 打印列表

```
>>> li = ['a', 'b', 'c']
>>> print "\n".join(li)
a
b
c
```

[1]	在你处理列表时，这确实是一个有用的调试技巧。在 Python 中，你会十分频繁地操作列表。

上述就是最后一个令人困惑的地方了。但是现在你应该已经理解这段代码了。

```
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])
```

4.9. 小结

`apihelper.py` 程序和它的输出现在应该非常清晰了。

```
def info(object, spacing=10, collapse=1):
    """Print methods and doc strings.
    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])
if __name__ == "__main__":
    print info.__doc__
```

`apihelper.py` 的输出：

```
>>> from apihelper import info
>>> li = []
>>> info(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

在研究下一章前，确保你可以无困难的完成下面这些事情：

- 用可选和命名参数定义和调用函数
- 用 `str` 强制转换任意值为字符串形式
- 用 `getattr` 动态得到函数和其它属性的引用
- 扩展列表解析语法实现列表过滤
- 识别 `and-or` 技巧并安全地使用它
- 定义 `lambda` 函数
- 将函数赋值给变量然后通过引用变量调用函数。我强调的已经够多了：这种思考方式对于提高对 Python 的理解力至关重要。在本书中你会随处可见这种技术的更复杂的应用。

第 5 章 对象和面向对象

这一章，和此后的许多章，均讨论了面向对象的 Python 程序设计。

5.1. 概览

下面是一个完整的，可运行的 Python 程序。请阅读模块、类和函数的 `doc string`s，可以大概了解这个程序所做的事情和工作情况。像平时一样，不用担心你不理解的东西，这就是本章其它部分将告诉你的内容。

例 5.1. `fileinfo.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
"""Framework for getting filetype-specific metadata.
Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key-value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])
Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...
Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict
def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename
class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}
    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass
    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
               for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]
if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]):
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

[1]	这个程序的输入要取决于你硬盘上的文件。为了得到有意义的输出，你应该修改目录路径指向你自已机器上的一个 MP3 文件目录。

下面就是从我的机器上得到的输出。你的输出将不一样，除非，由于某些令人吃惊的巧合，你与我有着共同的音乐品味。

```
album=
artist=Ghost in the Machine
title=A Time Long Forgotten (Concept
genre=31
name=/music/_singles/a_time_long_forgotten_con.mp3
year=1999
comment=http://mp3.com/ghostmachine
album=Rave Mix
artist=***DJ MARY-JANE***
title=HELLRAISER****Trance from Hell
genre=31
name=/music/_singles/hellraiser.mp3
year=2000
comment=http://mp3.com/DJMARYJANE
album=Rave Mix
artist=***DJ MARY-JANE***
title=KAIRO****THE BEST GOA
genre=31
name=/music/_singles/kairo.mp3
year=2000
comment=http://mp3.com/DJMARYJANE
album=Journeys
artist=Masters of Balance
title=Long Way Home
genre=31
name=/music/_singles/long_way_home1.mp3
year=2000
comment=http://mp3.com/MastersofBalan
album=
artist=The Cynic Project
title=Sidewinder
genre=18
name=/music/_singles/sidewinder.mp3
year=2000
comment=http://mp3.com/cynicproject
album=Digitosis@128k
artist=VXpanded
title=Spinning
genre=255
name=/music/_singles/spinning.mp3
year=2000
comment=http://mp3.com/artists/95/vxp
```

5.2. 使用 `from _module_ import` 导入模块

Python 有两种导入模块的方法。两种都有用，你应该知道什么时候使用哪一种方法。一种方法，`import _module_`，你已经在第 2.4 节 “万物皆对象”看过了。另一种方法完成同样的事情，但是它与第一种有着细微但重要的区别。

下面是 `from _module_ import` 的基本语法：

```
from UserDict import UserDict
```

它与你所熟知的 `import _module_` 语法很相似，但是有一个重要的区别：`UserDict` 被直接导入到局部名字空间去了，所以它可以直接使用，而不需要加上模块名的限定。你可以导入独立的项或使用 `from _module_ import *` 来导入所有东西。

> 注意
> Python 中的 `from _module_ import *` 像 Perl 中的 `use _module_` ；Python 中的 `import _module_` 像 Perl 中的 `require _module_` 。
>
> 注意
> Python 中的 `from _module_ import *` 像 Java 中的 `import _module_.*` ；Python 中的 `import _module_` 像 Java 中的 `import _module_` 。

例 5.2. `import _module_` *vs.* `from _module_ import`

```
>>> import types
>>> types.FunctionType
<type 'function'>
>>> FunctionType
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'FunctionType'
>>> from types import FunctionType
>>> FunctionType
<type 'function'>
```

[1]	`types` 模块不包含方法，只是表示每种 Python 对象类型的属性。注意这个属性必需用模块名 `types` 进行限定。
[2]	`FunctionType` 本身没有被定义在当前名字空间中；它只存在于 `types` 的上下文环境中。
[3]	这个语法从 `types` 模块中直接将 `FunctionType` 属性导入到局部名字空间中。
[4]	现在 `FunctionType` 可以直接使用，与 `types` 无关了。

什么时候你应该使用 `from _module_ import`？

- 如果你要经常访问模块的属性和方法，且不想一遍又一遍地敲入模块名，使用 `from _module_ import`。
- 如果你想要有选择地导入某些属性和方法，而不想要其它的，使用 `from _module_ import`。
- 如果模块包含的属性和方法与你的某个模块同名，你必须使用 `import _module_` 来避免名字冲突。

除了这些情况，剩下的只是风格问题了，你会看到用两种方式编写的 Python 代码。

> 小心
> 尽量少用 `from module import *`，因为判定一个特殊的函数或属性是从哪来的有些困难，并且会造成调试和重构都更困难。

进一步阅读关于模块导入技术

- eff-bot 有更多关于 `import _module_` *vs.* `from _module_ import` 的论述。
- *Python Tutorial* 讨论了高级的导入技术，包括 `from _module_ import *`。

5.3. 类的定义

- 5.3.1. 初始化并开始类编码
- 5.3.2. 了解何时去使用 self 和 **init**

Python 是完全面向对象的：你可以定义自已的类，从自已的或内置的类继承，然后从你定义的类创建实例。

在 Python 中定义类很简单。就像定义函数，没有单独的接口定义。只要定义类，然后就可以开始编码。Python 类以保留字 `class` 开始，后面跟着类名。从技术上讲，有这些就够了，因为一个类并非必须从其它类继承。

例 5.3. 最简单的 Python 类

```
class Loaf:
    pass
```

[1]	这个类的名字是 `Loaf`，它没有从其它类继承。类名通常是第一个字母大写，如：`EachWordLikeThis`，但这只是一个习惯，不是一个必要条件。
[2]	这个类没有定义任何方法或属性，但是从语法上，需要在定义中有些东西，所以你使用 `pass`。这是一个 Python 保留字，仅仅表示 “向前走，不要往这看”。它是一条什么都不做的语句，当你删空函数或类时，它是一个很好的占位符。
[3]	你可能猜到了，在类中的所有东西都要缩近，就像位于函数、`if` 语句，`for` 循环，诸如此类的代码。第一条不缩近的东西不属于这个类。

> 注意
> 在 Python 中的 `pass` 语句就像 Java 或 C 中的大括号空集 (`{}`)。

当然，实际上大多数的类都是从其它的类继承来的，并且它们会定义自已的类方法和属性。但是就像你刚才看到的，除了名字以外，类没有什么必须要具有的。特别是，C++ 程序员可能会感到奇怪，Python 的类没有显示的构造函数和析构函数。Python 类的确存在与构造函数相似的东西：`__init__` 方法。

例 5.4. 定义 `FileInfo` 类

```
 from UserDict import UserDict
class FileInfo(UserDict):
```

[1]	在 Python 中，类的基类只是简单地列在类名后面的小括号里。所以 `FileInfo` 类是从 `UserDict` 类 (它是从 `UserDict` 模块导进来的) 继承来的。 `UserDict` 是一个像字典一样工作的类，它允许你完全子类化字典数据类型，同时增加你自已的行为。{也存在相似的类 `UserList` 和 `UserString` ，它们允许你子类化列表和字符串。)[2] 在这个类的背后有一些“巫术”，我们将在本章的后面，随着更进一步地研究 `UserDict` 类，揭开这些秘密。

> 注意
> 在 Python 中，类的基类只是简单地列在类名后面的小括号里。不像在 Java 中有一个特殊的 `extends` 关键字。

Python 支持多重继承。在类名后面的小括号中，你可以列出许多你想要的类名，以逗号分隔。

5.3.1. 初始化并开始类编码

本例演示了使用 `__init__` 方法来进行 `FileInfo` 类的初始化。

例 5.5. 初始化 `FileInfo` 类

```
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
```

[1]	类也可以 (并且应该) 有 `doc string` s ，就像方法和函数一样。
[2]	`__init__` 在类的实例创建后被立即调用。它可能会引诱你称之为类的构造函数，但这种说法并不正确。说它引诱，是因为它看上去像 (按照习惯， `__init__` 是类中第一个定义的方法)，行为也像 (在一个新创建的类实例中，它是首先被执行的代码)，并且叫起来也像 (“init”当然意味着构造的本性)。说它不正确，是因为对象在调用 `__init__` 时已经被构造出来了，你已经有了一个对类的新实例的有效引用。但 `__init__` 是在 Python 中你可以得到的最接近构造函数的东西，并且它也扮演着非常相似的角色。
[3]	每个类方法的第一个参数，包括 `__init__` ，都是指向类的当前实例的引用。按照习惯这个参数总是被称为 `self` 。在 `__init__` 方法中， `self` 指向新创建的对象；在其它的类方法中，它指向方法被调用的类实例。尽管当定义方法时你需要明确指定 `self` ，但在调用方法时，你不 用指定它，Python 会替你自动加上的。
[4]	`__init__` 方法可以接受任意数目的参数，就像函数一样，参数可以用缺省值定义，即可以设置成对于调用者可选。在本例中， `filename` 有一个缺省值 `None` ，即 Python 的空值。

> 注意
> 习惯上，任何 Python 类方法的第一个参数 (对当前实例的引用) 都叫做 `self`。这个参数扮演着 C++ 或 Java 中的保留字 `this` 的角色，但 `self` 在 Python 中并不是一个保留字，它只是一个命名习惯。虽然如此，也请除了 `self` 之外不要使用其它的名字，这是一个非常坚固的习惯。

例 5.6. 编写 `FileInfo` 类

```
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename
```

[1]	一些伪面向对象语言，像 Powerbuilder 有一种“扩展”构造函数和其它事件的概念，即父类的方法在子类的方法执行前被自动调用。Python 不是这样，你必须显示地调用在父类中的合适方法。
[2]	我告诉过你，这个类像字典一样工作，那么这里就是第一个印象。我们将参数 `filename` 赋值给对象 `name` 关键字，作为它的值。
[3]	注意 `__init__` 方法从不返回一个值。

5.3.2. 了解何时去使用 `self` 和 `__init__`

当定义你自已的类方法时，你必须 明确将 `self` 作为每个方法的第一个参数列出，包括 `__init__`。当从你的类中调用一个父类的一个方法时，你必须包括 `self` 参数。但当你从类的外部调用你的类方法时，你不必对 `self` 参数指定任何值；你完全将其忽略，而 Python 会自动地替你增加实例的引用。我知道刚开始这有些混乱，它并不是自相矛盾的，因为它依靠于一个你还不了解的区别 (在绑定与非绑定方法之间)，故看上去是矛盾的。

噢。我知道有很多知识需要吸收，但是你要掌握它。所有的 Python 类以相同的方式工作，所以一旦你学会了一个，就是学会了全部。如果你忘了别的任何事，也要记住这件事，因为我认定它会让你出错：

> 注意
> `__init__` 方法是可选的，但是一旦你定义了，就必须记得显示调用父类的 `__init__` 方法 (如果它定义了的话)。这样更是正确的：无论何时子类想扩展父类的行为，后代方法必须在适当的时机，使用适当的参数，显式调用父类方法。

进一步阅读关于 Python 类

- *Learning to Program* 有优雅的类的介绍。

- *How to Think Like a Computer Scientist* 展示了如何使用类来实现复合数据类型模型。
- *Python Tutorial* 深入考虑了类、名字空间和继承。
- Python Knowledge Base 回答了关于类的常见问题。

Footnotes

[2] 在 2.2 之后已经可以从 dict、list 来派生子类了，关于这一点作者在后文也会提到。——译注

5.4. 类的实例化

- 5.4.1. 垃圾回收

在 Python 中对类进行实例化很直接。要对类进行实例化，只要调用类 (就好像它是一个函数)，传入定义在 `__init__` 方法中的参数。返回值将是新创建的对象。

例 5.7. 创建 `FileInfo` 实例

```
>>> import fileinfo
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3")
>>> f.__class__
<class fileinfo.FileInfo at 010EC204>
>>> f.__doc__
'store file metadata'
>>> f
{'name': '/music/_singles/kairo.mp3'}
```

[1]	你正在创建 `FileInfo` 类 (定义在 `fileinfo` 模块中) 的实例，并且将新创建的实例赋值给变量 `f` 。你传入了一个参数， `/music/_singles/kairo.mp3` ，它将最后作为在 `FileInfo` 中 `__init__` 方法中的 `filename` 参数。
[2]	每一个类的实例有一个内置属性， `__class__` ，它是对象的类。(注意这个表示包括了在我机器上的实例的物理地址，你的表示不会一样。)Java 程序员可能对 `Class` 类熟悉，这个类包含了像 `getName` 和 `getSuperclass` 之类用来得到一个对象元数据信息的方法。在 Python 中，这类元数据可以直接通过对象本身的属性，像 `__class__` 、 `__name__` 和 `__bases__` 来得到。
[3]	你可以像对函数或模块一样来访问实例的 `doc string` 。一个类的所有实例共享相同的 `doc string` 。
[4]	还记得什么时候 `__init__` 方法将它的 `filename` 参数赋给 `self["name"]` 吗？哦，答案在这。在创建类实例时你传入的参数被正确发送到 `__init__` 方法中 (当我们创建类实例时，我们所传递的参数被正确地发送给 `__init__` 方法 (随同一起传递的还有对象的引用， `self` ，它是由 Python 自动添加的)。

> 注意
>
> 在 Python 中，创建类的实例只要调用一个类，仿佛它是一个函数就行了。不像 C++ 或 Java 有一个明确的 `new` 操作符。

5.4.1. 垃圾回收

如果说创建一个新的实例是容易的，那么销毁它们甚至更容易。通常，不需要明确地释放实例，因为当指派给它们的变量超出作用域时，它们会被自动地释放。内存泄漏在 Python 中很少见。

例 5.8. 尝试实现内存泄漏

```
>>> def leakmem():
... f = fileinfo.FileInfo('/music/_singles/kairo.mp3')
...
>>> for i in range(100):
... leakmem()
```

[1]	每次 `leakmem` 函数被调用，你创建了 `FileInfo` 的一个实例，将其赋给变量 `f`，这个变量是函数内的一个局部变量。然后函数结束时没有释放 `f`，所以你可能认为有内存泄漏，但是你错了。当函数结束时，局部变量 `f` 超出了作用域。在这个地方，不再有任何对 `FileInfo` 新创建实例的引用 (因为除了 `f` 我们从未将其赋值给其它变量)，所以 Python 替我们销毁掉实例。
[2]	不管我们调用 `leakmem` 函数多少次，决不会泄漏内存，因为每一次，Python 将在从 `leakmem` 返回前销毁掉新创建的 `FileInfo` 类实例。

对于这种垃圾收集的方式，技术上的术语叫做“引用计数”。Python 维护着对每个实例的引用列表。在上面的例子中，只有一个 `FileInfo` 的实例引用：局部变量 `f`。当函数结束时，变量 `f` 超出作用域，所以引用计数降为 `0`，则 Python 自动销毁掉实例。

在 Python 的以前版本中，存在引用计数失败的情况，这样 Python 不能在后面进行清除。如果你创建两个实例，它们相互引用 (例如，双重链表，每一个结点有都一个指向列表中前一个和后一个结点的指针)，任一个实例都不会被自动销毁，因为 Python (正确) 认为对于每个实例都存在一个引用。Python 2.0 有一种额外的垃圾回收方式，叫做“标记后清除”，它足够聪明，可以正确地清除循环引用。

作为曾经读过哲学专业的一员，让我感到困惑的是，当没有人对事物进行观察时，它们就消失了，但是这确实是在 Python 中所发生的。通常，你可以完全忘记内存管理，让 Python 在后面进行清理。

进一步阅读

- *Python Library Reference* 总结了像 `__class__` 之类的内置属性。
- *Python Library Reference* 提供了 `gc` 模块的文档，此模块给予你对 Python 的垃圾回收的底层控制权。

5.5. 探索 `UserDict` ：一个封装类

如你所见， `FileInfo` 是一个有着像字典一样的行为方式的类。为了进一步揭示这一点，让我们看一看在 `UserDict` 模块中的 `UserDict` 类，它是我们的 `FileInfo` 类的父类。它没有什么特别的，也是用 Python 写的，并且保存在一个 `.py` 文件里，就像我们其他的代码。特别之处在于，它保存在你的 Python 安装目录的 `lib` 目录下。

> 提示
> 在 Windows 下的 ActivePython IDE 中，你可以快速打开在你的库路径中的任何模块，使用 File->Locate... (**Ctrl-L**)。

例 5.9. 定义 `UserDict` 类

```
class UserDict:
    def __init__(self, dict=None):
        self.data = {}
        if dict is not None: self.update(dict)
```

[1]	注意 `UserDict` 是一个基类，不是从任何其他类继承而来。
[2]	这就是我们在 `FileInfo` 类中进行了覆盖的 `__init__` 方法。注意这个父类的参数列表与子类不同。很好，每个子类可以拥有自已的参数集，只要使用正确的参数调用父类就可以了。这里父类有一个定义初始值的方法 (通过在 `dict` 参数中传入一个字典)，这一方法我们的 `FileInfo` 没有用上。
[3]	Python 支持数据属性 (在 Java 和 Powerbuilder 中叫做 “实例变量”，在 C++ 中叫 “数据成员”)，它是由某个特定的类实例所拥有的数据。在本例中，每个 `UserDict` 实例将拥有一个 `data` 数据属性。要从类外的代码引用这个属性，需要用实例的名字限定它， `_instance_.data` ，限定的方法与你用模块的名字来限定函数一样。要在类的内部引用一个数据属性，我们使用 `self` 作为限定符。习惯上，所有的数据属性都在 `__init__` 方法中初始化为有意义的值。然而，这并不是必须的，因为数据属性，像局部变量一样，当你首次赋给它值的时候突然产生。
[4]	`update` 方法是一个字典复制器：它把一个字典中的键和值全部拷贝到另一个字典。这个操作并不 事先清空目标字典，如果一些键在目标字典中已经存在，则它们将被覆盖，那些键名在目标字典中不存在的则不改变。应该把 `update` 看作是合并函数，而不是复制函数。
[5]	这个语法你可能以前没看过 (我还没有在这本书中的例子中用过它)。这是一条 `if` 语句，但是没有在下一行有一个缩近块，而只是在冒号后面，在同一行上有单条语句。这完全是合法的，它只是当你在一个块中仅有一条语句时的一个简写。(它就像在 C++ 中没有用大括号包括的单行语句。) 你可以用这种语法，或者可以在后面的行写下缩近代码，但是不能对同一个块同时用两种方式。

> 注意
> Java 和 Powerbuilder 支持通过参数列表的重载，也就是 一个类可以有同名的多个方法，但这些方法或者是参数个数不同，或者是参数的类型不同。其它语言 (最明显如 PL/SQL) 甚至支持通过参数名的重载，也就是 一个类可以有同名的多个方法，这些方法有相同类型，相同个数的参数，但参数名不同。Python 两种都不支持，总之是没有任何形式的函数重载。一个 `__init__` 方法就是一个 `__init__` 方法，不管它有什么样的参数。每个类只能有一个 `__init__` 方法，并且如果一个子类拥有一个 `__init__` 方法，它总是 覆盖父类的 `__init__` 方法，甚至子类可以用不同的参数列表来定义它。
>
> 注意
> Python 的原作者 Guido 是这样解释方法覆盖的：“子类可以覆盖父类中的方法。因为方法没有特殊的优先级设置，父类中的一个方法在调用同类中的另一方法时，可能其实调用到的却是一个子类中覆盖父类同名方法的方法。 (C++ 程序员可能会这样想：所有的 Python 方法都是虚函数。)”如果你不明白 (它令我颇感困惑)，不必在意。我想我要跳过它。[3]
>
> 小心
> 应该总是在 `__init__` 方法中给一个实例的所有数据属性赋予一个初始值。这样做将会节省你在后面调试的时间，不必为捕捉因使用未初始化 (也就是不存在) 的属性而导致的 `AttributeError` 异常费时费力。

例 5.10. `UserDict` 常规方法

```
    def clear(self): self.data.clear()
    def copy(self):
        if self.__class__ is UserDict:
            return UserDict(self.data)
        import copy
        return copy.copy(self)
    def keys(self): return self.data.keys()
    def items(self): return self.data.items()
    def values(self): return self.data.values()
```

[1]	`clear` 是一个普通的类方法，可以在任何时候被任何人公开调用。注意， `clear` 像所有的类方法一样 (常规的或专用的)，使用 `self` 作为它的第一个参数。(记住，当你调用方法时，不用包括 `self` ；这件事是 Python 替你做的。) 还应注意这个封装类的基本技术：将一个真正的字典 (`data`) 作为数据属性保存起来，定义所有真正字典所拥有的方法，并且将每个类方法重定向到真正字典上的相应方法。(你可能已经忘了，字典的 `clear` 方法删除它的所有关键字和关键字相应的值。)
[2]	真正字典的 `copy` 方法会返回一个新的字典，它是原始字典的原样的复制 (所有的键-值对都相同)。但是 `UserDict` 不能简单地重定向到 `self.data.copy` ，因为那个方法返回一个真正的字典，而我们想要的是返回同一个类的一个新的实例，就像是 `self` 。
[3]	我们使用 `__class__` 属性来查看 `self` 是否是一个 `UserDict`，如果是，太好了，因为我们知道如何拷贝一个 `UserDict` ：只要创建一个新的 `UserDict` ，并传给它真正的字典，这个字典已经存放在 `self.data` 中了。然后你立即返回这个新的 `UserDict`，你甚至于不需要在下面一行中使用 `import copy` 。
[4]	如果 `self.__class__` 不是 `UserDict` ，那么 `self` 一定是 `UserDict` 的某个子类 (如可能为 `FileInfo`)，生活总是存在意外。 `UserDict` 不知道如何生成它的子类的一个原样的拷贝，例如，有可能在子类中定义了其它的数据属性，所以我们只能完全复制它们，确定拷贝了它们的全部内容。幸运的是，Python 带了一个模块可以正确地完成这件事，它叫做 `copy` 。在这里我不想深入细节 (然而它是一个绝对酷的模块，你是否已经想到要自己研究它了呢？)。说 `copy` 能够拷贝任何 Python 对象就够了，这就是我们在这里用它的原因。
[5]	其余的方法是直截了当的重定向到 `self.data` 的内置函数上。

> 注意
> 在 Python 2.2 之前的版本中，你不可以直接子类化字符串、列表以及字典之类的内建数据类型。作为补偿，Python 提供封装类来模拟内建数据类型的行为，比如：`UserString`、 `UserList` 和 `UserDict`。通过混合使用普通和特殊方法，`UserDict` 类能十分出色地模仿字典。在 Python 2.2 和其后的版本中，你可以直接从 `dict` 内建数据类型继承。本书 `fileinfo_fromdict.py` 中有这方面的一个例子。

如例子中所示，在 Python 中，你可以直接继承自内建数据类型 `dict`，这样做有三点与 `UserDict` 不同。

例 5.11. 直接继承自内建数据类型 `dict`

```
class FileInfo(dict):
    "store file metadata"
    def __init__(self, filename=None):
        self["name"] = filename
```

[1]	第一个区别是你不需要导入 `UserDict` 模块，因为 `dict` 是已经可以使用的内建数据类型。第二个区别是你不是继承自 `UserDict.UserDict` ，而是直接继承自 `dict` 。
[2]	第三个区别有些晦涩，但却很重要。 `UserDict` 内部的工作方式要求你手工地调用它的 `__init__` 方法去正确初始化它的内部数据结构。 `dict` 并不这样工作，它不是一个封装所以不需要明确的初始化。

进一步阅读

- *Python Library Reference* 提供了 `UserDict` 模块 和 `copy` 模块 的文档。

Footnotes

[3] 实际上，这一点并不是那么难以理解。考虑两个类， `base` 和 `child` ， `base` 中的方法 `a` 需要调用 `self.b` ；而我们又在 `child` 中覆盖了方法 `b` 。然后我们创建一个 `child` 的实例， `ch` 。调用 `ch.a` ，那么此时的方法 `a` 调用的 `b` 函数将不是 `base.b` ，而是 `child.b` 。——译注

5.6. 专用类方法

- 5.6.1. 获得和设置数据项

除了普通的类方法，Python 类还可以定义专用方法。专用方法是在特殊情况下或当使用特别语法时由 Python 替你调用的，而不是在代码中直接调用 (像普通的方法那样)。

就像你在上一节所看到的，普通的方法对在类中封装字典很有帮助。但是只有普通方法是不够的，因为除了对字典调用方法之外，还有很多事情可以做的。例如，你可以通过一种没有包括明确方法调用的语法来获得和设置数据项。这就是专用方法产生的原因：它们提供了一种方法，可以将非方法调用语法映射到方法调用上。

5.6.1. 获得和设置数据项

例 5.12. `__getitem__` 专用方法

```
    def __getitem__(self, key): return self.data[key]
```

```
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3")
>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__getitem__("name")
'/music/_singles/kairo.mp3'
>>> f["name"]
'/music/_singles/kairo.mp3'
```

[1]	`__getitem__` 专用方法很简单。像普通的方法 `clear`，`keys` 和 `values` 一样，它只是重定向到字典，返回字典的值。但是怎么调用它呢？哦，你可以直接调用 `__getitem__`，但是在实际中你其实不会那样做：我在这里执行它只是要告诉你它是如何工作的。正确地使用 `__getitem__` 的方法是让 Python 来替你调用。
[2]	这个看上去就像你用来得到一个字典值的语法，事实上它返回你期望的值。下面是隐藏起来的一个环节：暗地里，Python 已经将这个语法转化为 `f.__getitem__("name")` 的方法调用。这就是为什么 `__getitem__` 是一个专用类方法的原因，不仅仅是你可以自已调用它，还可以通过使用正确的语法让 Python 来替你调用。

当然，Python 有一个与 `__getitem__` 类似的 `__setitem__` 专用方法，参见下面的例子。

例 5.13. `__setitem__` 专用方法

```
    def __setitem__(self, key, item): self.data[key] = item
```

```
>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__setitem__("genre", 31)
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':31}
>>> f["genre"] = 32
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':32}
```

[1]	与 `__getitem__` 方法一样， `__setitem__` 简单地重定向到真正的字典 `self.data` ，让它来进行工作。并且像 `__getitem__` 一样，通常你不会直接调用它，当你使用了正确的语法，Python 会替你调用 `__setitem__` 。
[2]	这个看上去像正常的字典语法，当然除了 `f` 实际上是一个类，它尽可能地打扮成一个字典，并且 `__setitem__` 是打扮的一个重点。这行代码实际上暗地里调用了 `f.__setitem__("genre", 32)` 。

`__setitem__` 是一个专用类方法，因为它可以让 Python 来替你调用，但是它仍然是一个类方法。就像在 `UserDict` 中定义 `__setitem__` 方法一样容易，我们可以在子类中重新定义它，对父类的方法进行覆盖。这就允许我们定义出在某些方面像字典一样动作的类，但是可以定义它自已的行为，超过和超出内置的字典。

这个概念是本章中我们正在学习的整个框架的基础。每个文件类型可以拥有一个处理器类，这些类知道如何从一个特殊的文类型得到元数据。只要知道了某些属性 (像文件名和位置)，处理器类就知道如何自动地得到其它的属性。它的实现是通过覆盖 `__setitem__` 方法，检查特别的关键字，然后当找到后加入额外的处理。

例如， `MP3FileInfo` 是 `FileInfo` 的子类。在设置了一个 `MP3FileInfo` 类的 `name` 时，并不只是设置 `name` 关键字 (像父类 `FileInfo` 所做的)，它还要在文件自身内进行搜索 MP3 的标记然后填充一整套关键字。下面的例子将展示其工作方式。

例 5.14. 在 `MP3FileInfo` 中覆盖 `__setitem__`

```
    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)
```

[1]	注意我们的 `__setitem__` 方法严格按照与父类方法相同的形式进行定义。这一点很重要，因为 Python 将替你执行方法，而它希望这个函数用确定个数的参数进行定义。(从技术上说，参数的名字没有关系，只是个数。)
[2]	这里就是整个 `MP3FileInfo` 类的难点：如果给 `name` 关键字赋一个值，我们还想做些额外的事情。
[3]	我们对 `name` 所做的额外处理封装在了 `__parse` 方法中。这是定义在 `MP3FileInfo` 中的另一个类方法，则当我们调用它时，我们用 `self` 对其限定。仅是调用 `__parse` 将只会看成定义在类外的普通方法，调用 `self.__parse` 将会看成定义在类中的一个类方法。这不是什么新东西，你用同样的方法来引用数据属性。
[4]	在做完我们额外的处理之后，我们需要调用父类的方法。记住，在 Python 中不会自动为你完成，需手工执行。注意，我们在调用直接父类， `FileInfo` ，尽管它没有 `__setitem__` 方法。没问题，因为 Python 将会沿着父类树走，直到它找到一个拥有我们正在调用方法的类，所以这行代码最终会找到并且调用定义在 `UserDict` 中的 `__setitem__` 。

> 注意
> 当在一个类中存取数据属性时，你需要限定属性名：`self._attribute_`。当调用类中的其它方法时，你属要限定方法名：`self._method_`。

例 5.15. 设置 `MP3FileInfo` 的 `name`

```
>>> import fileinfo
>>> mp3file = fileinfo.MP3FileInfo()
>>> mp3file
{'name':None}
>>> mp3file["name"] = "/music/_singles/kairo.mp3"
>>> mp3file
{'album': 'Rave Mix', 'artist': '***DJ MARY-JANE***', 'genre': 31,
'title': 'KAIRO****THE BEST GOA', 'name': '/music/_singles/kairo.mp3',
'year': '2000', 'comment': 'http://mp3.com/DJMARYJANE'}
>>> mp3file["name"] = "/music/_singles/sidewinder.mp3"
>>> mp3file
{'album': '', 'artist': 'The Cynic Project', 'genre': 18, 'title': 'Sidewinder',
'name': '/music/_singles/sidewinder.mp3', 'year': '2000',
'comment': 'http://mp3.com/cynicproject'}
```

[1]	首先，我们创建了一个 `MP3FileInfo` 的实例，没有传递给它文件名。(我们可以不用它，因为 `__init__` 方法的 `filename` 参数是可选的。) 因为 `MP3FileInfo` 没有它自己的 `__init__` 方法，Python 沿着父类树走，发现了 `FileInfo` 的 `__init__` 方法。这个 `__init__` 方法手工调用了 `UserDict` 的 `__init__` 方法，然后设置 `name` 关键字为 `filename`，它为 `None`，因为我们还没有传入一个文件名。所以，`mp3file` 最初看上去像是有一个关键字的字典，`name` 的值为 `None`。
[2]	现在真正有趣的开始了。设置 `mp3file` 的 `name` 关键字触发了 `MP3FileInfo` 上的 `__setitem__` 方法 (而不是 `UserDict` 的)，这个方法注意到我们正在用一个真实的值来设置 `name` 关键字，接着调用 `self.__parse`。尽管我们完全还没有研究过 `__parse` 方法，从它的输出你可以看出，它设置了其它几个关键字：`album`、`artist`、`genre`、`title`、`year` 和 `comment`。
[3]	修改 `name` 关键字将再次经受同样的处理过程：Python 调用 `__setitem__`，`__setitem__` 调用 `self.__parse`，`self.__parse` 设置其它所有的关键字。

5.7. 高级专用类方法

除了 `__getitem__` 和 `__setitem__` 之外 Python 还有更多的专用函数。某些可以让你模拟出你甚至可能不知道的功能。

下面的例子将展示 `UserDict` 一些其他专用方法。

例 5.16. `UserDict` 中更多的专用方法

```
    def __repr__(self): return repr(self.data)
    def __cmp__(self, dict):
        if isinstance(dict, UserDict):
            return cmp(self.data, dict.data)
        else:
            return cmp(self.data, dict)
    def __len__(self): return len(self.data)
    def __delitem__(self, key): del self.data[key]
```

[1]	`__repr__` 是一个专用的方法，在当调用 `repr(_instance_)` 时被调用。`repr` 函数是一个内置函数，它返回一个对象的字符串表示。它可以用在任何对象上，不仅仅是类的实例。你已经对 `repr` 相当熟悉了，尽管你不知道它。在交互式窗口中，当你只敲入一个变量名，接着按**ENTER**，Python 使用 `repr` 来显示变量的值。自已用一些数据来创建一个字典 `d` ，然后用 `print repr(d)` 来看一看吧。
[2]	`__cmp__` 在比较类实例时被调用。通常，你可以通过使用 `==` 比较任意两个 Python 对象，不只是类实例。有一些规则，定义了何时内置数据类型被认为是相等的，例如，字典在有着全部相同的关键字和值时是相等的。对于类实例，你可以定义 `__cmp__` 方法，自已编写比较逻辑，然后你可以使用 `==` 来比较你的类，Python 将会替你调用你的 `__cmp__` 专用方法。
[3]	`__len__` 在调用 `len(_instance_)` 时被调用。`len` 是一个内置函数，可以返回一个对象的长度。它可以用于任何被认为理应有长度的对象。字符串的 `len` 是它的字符个数；字典的 `len` 是它的关键字的个数；列表或序列的 `len` 是元素的个数。对于类实例，定义 `__len__` 方法，接着自已编写长度的计算，然后调用 `len(_instance_)`，Python 将替你调用你的 `__len__` 专用方法。
[4]	`__delitem__` 在调用 `del _instance_[_key_]` 时调用，你可能记得它作为从字典中删除单个元素的方法。当你在类实例中使用 `del` 时，Python 替你调用 `__delitem__` 专用方法。

> 注意
> 在 Java 中，通过使用 `str1 == str2` 可以确定两个字符串变量是否指向同一块物理内存位置。这叫做对象同一性，在 Python 中写为 `str1 is str2` 。在 Java 中要比较两个字符串值，你要使用 `str1.equals(str2)` ；在 Python 中，你要使用 `str1 == str2` 。某些 Java 程序员，他们已经被教授得认为，正是因为在 Java 中 `==` 是通过同一性而不是值进行比较，所以世界才会更美好。这些人要接受 Python 的这个“严重缺失”可能要花些时间。

在这个地方，你可能会想，“所有这些工作只是为了在类中做一些我可以对一个内置数据类型所做的操作”。不错，如果你能够从像字典一样的内置数据类型进行继承的话，事情就容易多了 (那样整个 `UserDict` 类将完全不需要了)。尽管你可以这样做，专用方法仍然是有用的，因为它们可以用于任何的类，而不只是像 `UserDict` 这样的封装类。

专用方法意味着任何类 可以像字典一样保存键-值对，只要定义 `__setitem__` 方法。任何类可以表现得像一个序列，只要定义 `__getitem__` 方法。任何定义了 `__cmp__` 方法的类可以用 `==` 进行比较。并且如果你的类表现为拥有类似长度的东西，不要定义 `GetLength` 方法，而定义 `__len__` 方法，并使用 `len(_instance_)` 。

> 注意
> 其它的面向对象语言仅让你定义一个对象的物理模型 (“这个对象有 `GetLength` 方法”)，而 Python 的专用类方法像 `__len__` 允许你定义一个对象的逻辑模型 (“这个对象有一个长度”)。

Python 存在许多其它的专用方法。有一整套的专用方法，可以让类表现得象数值一样，允许你在类实例上进行加、减，以及执行其它算数操作。(关于这一点典型的例子就是表示复数的类，数值带有实数和虚数部分。) `__call__` 方法让一个类表现得像一个函数，允许你直接调用一个类实例。并且存在其它的专用函数，允许类拥有只读或只写数据属性，在后面的章节中我们会更多地谈到这些。

进一步阅读

- *Python Reference Manual* 提供了所有专用类方法的文档。

5.8. 类属性介绍

你已经知道了数据属性，它们是被一个特定的类实例所拥有的变量。Python 也支持类属性，它们是由类本身所拥有的。

例 5.17. 类属性介绍

```
class MP3FileInfo(FileInfo):
   "store ID3v1.0 MP3 tags"
   tagDataMap = {"title"   : (  3,  33, stripnulls),
                 "artist"  : ( 33,  63, stripnulls),
                 "album"   : ( 63,  93, stripnulls),
                 "year"    : ( 93,  97, stripnulls),
                 "comment" : ( 97, 126, stripnulls),
                 "genre"   : (127, 128, ord)}
```

```
>>> import fileinfo
>>> fileinfo.MP3FileInfo
<class fileinfo.MP3FileInfo at 01257FDC>
>>> fileinfo.MP3FileInfo.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
>>> m = fileinfo.MP3FileInfo()
>>> m.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
```

[1]	`MP3FileInfo` 是类本身，不是任何类的特别实例。
[2]	`tagDataMap` 是一个类属性：字面的意思，一个类的属性。它在创建任何类实例之前就有效了。
[3]	类属性既可以通过直接对类的引用，也可以通过对类的任意实例的引用来使用。

> 注意
> 在 Java 中，静态变量 (在 Python 中叫类属性) 和实例变量 (在 Python 中叫数据属性) 两者都是紧跟在类定义之后定义的 (一个有 `static` 关键字，一个没有)。在 Python 中，只有类属性可以定义在这里，数据属性定义在 `__init__` 方法中。

类属性可以作为类级别的常量来使用 (这就是为什么我们在 `MP3FileInfo` 中使用它们)，但是它们不是真正的常量。你也可以修改它们。

> 注意
> 在 Python 中没有常量。如果你试图努力的话什么都可以改变。这一点满足 Python 的核心原则之一：坏的行为应该被克服而不是被取缔。如果你真正想改变 `None` 的值，也可以做到，但当无法调试的时候别来找我。

例 5.18. 修改类属性

```
>>> class counter:
... count = 0
... def __init__(self):
...         self.__class__.count += 1
...
>>> counter
<class __main__.counter at 010EAECC>
>>> counter.count
0
>>> c = counter()
>>> c.count
1
>>> counter.count
1
>>> d = counter()
>>> d.count
2
>>> c.count
2
>>> counter.count
2
```

[1]	`count` 是 `counter` 类的一个类属性。
[2]	`__class__` 是每个类实例的一个内置属性 (也是每个类的)。它是一个类的引用，而 `self` 是一个类 (在本例中，是 `counter` 类) 的实例。
[3]	因为 `count` 是一个类属性，它可以在我们创建任何类实例之前，通过直接对类引用而得到。
[4]	创建一个类实例会调用 `__init__` 方法，它会给类属性 `count` 加 `1` 。这样会影响到类自身，不只是新创建的实例。
[5]	创建第二个实例将再次增加类属性 `count` 。注意类属性是如何被类和所有类实例所共享的。

5.9. 私有函数

与大多数语言一样，Python 也有私有的概念：

- 私有函数不可以从它们的模块外面被调用
- 私有类方法不能够从它们的类外面被调用
- 私有属性不能够从它们的类外面被访问

与大多数的语言不同，一个 Python 函数，方法，或属性是私有还是公有，完全取决于它的名字。

如果一个 Python 函数，类方法，或属性的名字以两个下划线开始 (但不是结束)，它是私有的；其它所有的都是公有的。 Python 没有类方法保护 的概念 (只能用于它们自已的类和子类中)。类方法或者是私有 (只能在它们自已的类中使用) 或者是公有 (任何地方都可使用)。

在 `MP3FileInfo` 中，有两个方法：`__parse` 和 `__setitem__`。正如我们已经讨论过的，`__setitem__` 是一个专有方法；通常，你不直接调用它，而是通过在一个类上使用字典语法来调用，但它是公有的，并且如果有一个真正好的理由，你可以直接调用它 (甚至从 `fileinfo` 模块的外面)。然而，`__parse` 是私有的，因为在它的名字前面有两个下划线。

> 注意
> 在 Python 中，所有的专用方法 (像 `__setitem__`) 和内置属性 (像 `__doc__`) 遵守一个标准的命名习惯：开始和结束都有两个下划线。不要对你自已的方法和属性用这种方法命名；到最后，它只会把你 (或其它人) 搞乱。

例 5.19. 尝试调用一个私有方法

```
>>> import fileinfo
>>> m = fileinfo.MP3FileInfo()
>>> m.__parse("/music/_singles/kairo.mp3")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'MP3FileInfo' instance has no attribute '__parse'
```

[1]	如果你试图调用一个私有方法，Python 将引发一个有些误导的异常，宣称那个方法不存在。当然它确实存在，但是它是私有的，所以在类外是不可使用的。严格地说，私有方法在它们的类外是可以访问的，只是不容易 处理。在 Python 中没有什么是真正私有的；在内部，私有方法和属性的名字被忽然改变和恢复，以致于使得它们看上去用它们给定的名字是无法使用的。你可以通过 `_MP3FileInfo__parse` 名字来使用 `MP3FileInfo` 类的 `__parse` 方法。知道了这个方法很有趣，然后要保证决不在真正的代码中使用它。私有方法由于某种原因而私有，但是像其它很多在 Python 中的东西一样，它们的私有化基本上是习惯问题，而不是强迫的。

进一步阅读

- *Python Tutorial* 讨论了私有变量的内部工作方式。

5.10. 小结

实打实的对象把戏到此为止。你将在 第 12 章 中看到一个真实世界应用程序的专有类方法，它使用 `getattr` 创建一个到远程 Web 服务的代理。

下一章将继续使用本章的例程探索其他 Python 的概念，例如：异常、文件对象 和 `for` 循环。

在研究下一章之前，确保你可以无困难地完成下面的事情：

- 使用 `import _module_` 或 `from _module_ import` 导入模块
- 定义和实例化类
- 定义 `__init__` 方法和其他专用类方法，并理解它们何时会调用
- 子类化 `UserDict` 来定义行为像字典的类
- 定义数据属性和类属性，并理解它们之间的不同
- 定义私有属性和方法

第 6 章 异常和文件处理

在本章中，将研究异常、文件对象、 `for` 循环、 `os` 和 `sys` 模块等内容。如果你已经在其它编程语言中使用过异常，你可以简单看看第一部分来了解 Python 的语法。但是本章其它的内容仍需仔细研读。

6.1. 异常处理

- 6.1.1. 为其他用途使用异常

与许多面向对象语言一样，Python 具有异常处理，通过使用 `try...except` 块来实现。

> 注意
> Python 使用 `try...except` 来处理异常，使用 `raise` 来引发异常。Java 和 C++ 使用 `try...catch` 来处理异常，使用 `throw` 来引发异常。

异常在 Python 中无处不在；实际上在标准 Python 库中的每个模块都使用了它们，并且 Python 自已会在许多不同的情况下引发它们。在整本书中你已经再三看到它们了。

- 使用不存在的字典关键字将引发 `KeyError` 异常。
- 搜索列表中不存在的值将引发 `ValueError` 异常。
- 调用不存在的方法将引发 `AttributeError` 异常。
- 引用不存在的变量将引发 `NameError` 异常。
- 未强制转换就混用数据类型将引发 `TypeError` 异常。

在这些情况下，我们都在简单地使用 Python IDE：一个错误发生了，异常被打印出来 (取决于你的 IDE，可能会有意地以一种刺眼的红色形式表示)，这便是。这叫做未处理 异常；当异常被引发时，没有代码来明确地关注和处理它，所以异常被传给置在 Python 中的缺省的处理，它会输出一些调试信息并且终止运行。在 IDE 中，这不是什么大事，但是如果发生在你真正的 Python 程序运行的时候，整个程序将会终止。

然而，一个异常不一定会引起程序的完全崩溃。当异常引发时，可以被处理 掉。有时候一个异常实际是因为代码中的 bug (比如使用一个不存在的变量)，但是许多时候，一个异常是可以预见的。如果你打开一个文件，它可能不存在。如果你连接一个数据库，它可能不可连接或没有访问所需的正确的安全证书。如果知道一行代码可能会引发异常，你应该使用一个 `try...except` 块来处理异常。

例 6.1. 打开一个不存在的文件

```
>>> fsock = open("/notthere", "r")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: '/notthere'
>>> try:
... fsock = open("/notthere")
... except IOError:
... print "The file does not exist, exiting gracefully"
... print "This line will always print"
The file does not exist, exiting gracefully
This line will always print
```

[1]	使用内置 `open` 函数，我们可以试着打开一个文件来读取 (在下一节有关于 `open` 的更多内容)。但是那个文件不存在，所以这样就引发 `IOError` 异常。因为我们没有提供任何显式的对 `IOError` 异常的检查，Python 仅仅打印出某个关于发生了什么的调试信息，然后终止。
[2]	我们试图打开同样不存在的文件，但是这次我们在一个 `try...except` 内来执行它。
[3]	当 `open` 方法引发 `IOError` 异常时，我们已经准备好处理它了。`except IOError:` 行捕捉异常，接着执行我们自己的代码块，这个代码块在本例中只是打印出更令人愉快的错误信息。
[4]	一旦异常被处理了，处理通常在 `try...except` 块之后的第一行继续进行。注意这一行将总是打印出来，无论异常是否发生。如果在你的根目录下确实有一个叫 `notthere` 的文件，对 `open` 的调用将成功， `except` 子句将忽略，并且最后一行仍将执行。

异常可能看上去不友好 (毕竟，如果你不捕捉异常，整个程序将崩溃)，但是考虑一下别的方法。你该不会希望获得一个指向不存在的文件的对象吧？不管怎么样你都得检查它的有效性，而且如果你忘记了，你的程序将会在下面某个地方给出奇怪的错误，这样你将不得不追溯到源程序。我确信你做过这种事；这可并不有趣。使用异常，一发生错误，你就可以在问题的源头通过标准的方法来处理它们。

6.1.1. 为其他用途使用异常

除了处理实际的错误条件之外，对于异常还有许多其它的用处。在标准 Python 库中一个普通的用法就是试着导入一个模块，然后检查是否它能使用。导入一个并不存在的模块将引发一个 `ImportError` 异常。你可以使用这种方法来定义多级别的功能——依靠在运行时哪个模块是有效的，或支持多种平台 (即平台特定代码被分离到不同的模块中)。

你也能通过创建一个从内置的 `Exception` 类继承的类定义你自己的异常，然后使用 `raise` 命令引发你的异常。如果你对此感兴趣，请看进一步阅读的部分。

下面的例子演示了如何使用异常支持特定平台功能。代码来自 `getpass` 模块，一个从用户获得口令的封装模块。获得口令在 UNIX、Windows 和 Mac OS 平台上的实现是不同的，但是这个代码封装了所有的不同之处。

例 6.2. 支持特定平台功能

```
# Bind the name getpass to the appropriate function
try:
    import termios, TERMIOS
except ImportError:
    try:
        import msvcrt
    except ImportError:
        try:
            from EasyDialogs import AskPassword
        except ImportError:
            getpass = default_getpass
        else:
            getpass = AskPassword
    else:
        getpass = win_getpass
else:
    getpass = unix_getpass
```

[1]	`termios` 是 UNIX 独有的一个模块，它提供了对于输入终端的底层控制。如果这个模块无效 (因为它不在你的系统上，或你的系统不支持它)，则导入失败，Python 引发我们捕捉的 `ImportError` 异常。
[2]	OK，我们没有 `termios` ，所以让我们试试 `msvcrt` ，它是 Windows 独有的一个模块，可以提供在 Microsoft Visual C++ 运行服务中的许多有用的函数的一个API。如果导入失败，Python 会引发我们捕捉的 `ImportError` 异常。
[3]	如果前两个不能工作，我们试着从 `EasyDialogs` 导入一个函数，它是 Mac OS 独有的一个模块，提供了各种各样类型的弹出对话框。再一次，如果导入失败，Python 会引发一个我们捕捉的 `ImportError` 异常。
[4]	这些平台特定的模块没有一个有效 (有可能，因为 Python 已经移植到了许多不同的平台上了)，所以我们需要回头使用一个缺省口令输入函数 (这个函数定义在 `getpass` 模块中的别的地方)。注意我们在这里所做的：我们将函数 `default_getpass` 赋给变量 `getpass` 。如果你读了官方 `getpass` 文档，它会告诉你 `getpass` 模块定义了一个 `getpass` 函数。它是这样做的：通过绑定 `getpass` 到正确的函数来适应你的平台。然后当你调用 `getpass` 函数时，你实际上调用了平台特定的函数，是这段代码已经为你设置好的。你不需要知道或关心你的代码正运行在何种平台上；只要调用 `getpass` ，则它总能正确处理。
[5]	一个 `try...except` 块可以有一条 `else` 子句，就像 `if` 语句。如果在 `try` 块中没有异常引发，然后 `else` 子句被执行。在本例中，那就意味着如果 `from EasyDialogs import AskPassword` 导入可工作，所以我们应该绑定 `getpass` 到 `AskPassword` 函数。其它每个 `try...except` 块有着相似的 `else` 子句，当我们发现一个 `import` 可用时，就绑定 `getpass` 到适合的函数。

进一步阅读

- *Python Tutorial* 讨论了异常，包括定义和引发你自己的异常，以及一次处理多个异常。
- *Python Library Reference* 总结了所有内置异常。
- *Python Library Reference* 提供了 getpass 模块的文档。
- *Python Library Reference* 提供了 `traceback` 模块 的文档，这个模块在异常引发之后，提供了底层的对异常属性的处理。

- *Python Reference Manual* 讨论了 `try...except` 块的内部工作方式。

6.2. 与文件对象共事

Python 有一个内置函数，`open`，用来打开在磁盘上的文件。`open` 返回一个文件对象，它拥有一些方法和属性，可以得到被打开文件的信息，以及对被打开文件进行操作。

例 6.3. 打开文件

```
>>> f = open("/music/_singles/kairo.mp3", "rb")
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.mode
'rb'
>>> f.name
'/music/_singles/kairo.mp3'
```

[1]	`open` 方法可以接收三个参数：文件名、模式和缓冲区参数。只有第一个参数 (文件名) 是必须的；其它两个是可选的。如果没有指定，文件以文本方式打开。这里我们以二进制方式打开文件进行读取。(`print open.__doc__` 会给出所有可能模式的很好的解释。)
[2]	`open` 函数返回一个对象 (到现在为止，这一点应该不会使你感到吃惊)。一个文件对象有几个有用的属性。
[3]	文件对象的 `mode` 属性告诉你文件以何种模式被打开。
[4]	文件对象的 `name` 属性告诉你文件对象所打开的文件名。

6.2.1. 读取文件

你打开文件之后，你要做的第一件事是从中读取，正如下一个例子所展示的。

例 6.4. 读取文件

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.tell()
0
>>> f.seek(-128, 2)
>>> f.tell()
7542909
>>> tagData = f.read(128)
>>> tagData
'TAGKAIRO****THE BEST GOA         ***DJ MARY-JANE***
Rave Mix                      2000http://mp3.com/DJMARYJANE     \037'
>>> f.tell()
7543037
```

[1]	一个文件对象维护它所打开文件的状态。文件对象的 `tell` 方法告诉你在被打开文件中的当前位置。因为我们还没有对这个文件做任何事，当前位置为 `0`，它是文件的起始处。
[2]	文件对象的 `seek` 方法在被打开文件中移动到另一个位置。第二个参数指出第一个参数是什么意思：`0` 表示移动到一个绝对位置 (从文件起始处算起)，`1` 表示移到一个相对位置 (从当前位置算起)，还有 `2` 表示相对于文件尾的位置。因为我们搜索的 MP3 标记保存在文件的末尾，我们使用 `2` 并且告诉文件对象从文件尾移动到 `128` 字节的位置。
[3]	`tell` 方法确认了当前位置已经移动了。
[4]	`read` 方法从被打开文件中读取指定个数的字节，并且返回含有读取数据的字符串。可选参数指定了读取的最大字节数。如果没有指定参数，`read` 将读到文件末尾。(我们本可以在这里简单地说 `read()`，因为我们确切地知道在文件的何处，事实上，我们读的是最后 128 个字节。) 读出的数据赋给变量 `tagData`，并且当前的位置根据所读的字节数作了修改。
[5]	`tell` 方法确认了当前位置已经移动了。如果做一下算术，你会看到在读了 128 个字节之后，位置数已经增加了 128。

6.2.2. 关闭文件

打开文件消耗系统资源，并且其间其它程序可能无法访问它们 (取决于文件模式)。这就是一旦操作完毕就该关闭文件的重要所在。

例 6.5. 关闭文件

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed
False
>>> f.close()
>>> f
<closed file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed
True
>>> f.seek(0)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.tell()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.read()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.close()
```

[1]	文件对象的 `closed` 属性表示对象是打开还是关闭了文件。在本例中，文件仍然打开着 (`closed` 是 `False`)。
[2]	为了关闭文件，调用文件对象的 `close` 方法。这样就释放掉你加在文件上的锁 (如果有的话)，刷新被缓冲的系统还未写入的输出 (如果有的话)，并且释放系统资源。
[3]	`closed` 属性证实了文件被关闭了。
[4]	文件被关闭了，但这并不意味着文件对象不再存在。变量 `f` 将继续存在，直到它超出作用域或被手工删除。然而，一旦文件被关闭，操作它的方法就没有一个能使用；它们都会引发异常。
[5]	对一个文件已经关闭的文件对象调用 `close` 不会 引发异常，它静静地失败。

6.2.3. 处理 I/O 错误

现在你已经足能理解前一章的例子程序 `fileinfo.py` 的文件处理代码了。下面这个例子展示了如何安全地打开文件和读取文件，以及优美地处理错误。

例 6.6. `MP3FileInfo` 中的文件对象

```
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            .
            .
            .
        except IOError:
            pass
```

[1]	因为打开和读取文件有风险，并且可能引发异常，所有这些代码都用一个 `try...except` 块封装。(嘿，标准化的缩近不好吗？这就是你开始欣赏它的地方。)
[2]	`open` 函数可能引发 `IOError` 异常。(可能是文件不存在。)
[3]	`seek` 方法可能引发 `IOError` 异常。(可能是文件长度小于 128 字节。)
[4]	`read` 方法可能引发 `IOError` 异常。(可能磁盘有坏扇区，或它在一个网络驱动器上，而网络刚好断了。)
[5]	这是新的：一个 `try...finally` 块。一旦文件通过 `open` 函数被成功地打开，我们应该绝对保证把它关闭，即使是在 `seek` 或 `read` 方法引发了一个异常时。`try...finally` 块可以用来：在 `finally` 块中的代码将总是 被执行，甚至某些东西在 `try` 块中引发一个异常也会执行。可以这样考虑，不管在路上发生什么，代码都会被 “即将灭亡” 地执行。
[6]	最后，处理我们的 `IOError` 异常。它可能是由调用 `open`、`seek` 或 `read` 引发的 `IOError` 异常。这里，我们其实不用关心，因为将要做的事就是静静地忽略它然后继续。(记住，`pass` 是一条不做任何事的 Python 语句。) 这样完全合法，“处理” 一个异常可以明确表示不做任何事。它仍然被认为处理过了，并且处理将正常继续，从 `try...except` 块的下一行代码开始。

6.2.4. 写入文件

正如你所期待的，你也能用与读取文件同样的方式写入文件。有两种基本的文件模式：

- 追加 (Append) 模式将数据追加到文件尾。
- 写入 (write) 模式将覆盖文件的原有内容。

如果文件还不存在，任意一种模式都将自动创建文件，因此从来不需要任何复杂的逻辑：“如果 log 文件还不存在，将创建一个新的空文件，正因为如此，你可以第一次就打开它”。打开文件并开始写就可以了。

例 6.7. 写入文件

```
>>> logfile = open('test.log', 'w')
>>> logfile.write('test succeeded')
>>> logfile.close()
>>> print file('test.log').read()
test succeeded
>>> logfile = open('test.log', 'a')
>>> logfile.write('line 2')
>>> logfile.close()
>>> print file('test.log').read()
test succeededline 2
```

[1]	你可以大胆地开始创建新文件 `test.log` 或覆盖现有文件，并为写入目的而打开它。(第二个参数 `"w"` 的意思是为文件写入而打开。) 是的，它和想象中的一样危险。我希望你不要关心文件以前的内容，因为它现在已经不存在了。
[2]	你可以使用 `open` 返回的文件对象的 `write` 方法向一个新打开的文件添加数据。
[3]	`file` 是 `open` 的同义语。这一行语句打开文件，读取内容，并打印它们。
[4]	碰巧你知道 `test.log` 存在 (因为你刚向它写完了数据)，所以你可以打开它并向其追加数据。(`"a"` 参数的意思是为追加目的打开文件。) 实际上即使文件不存在你也可以这样做，因为以追加方式打开一文件时，如果需要的话会创建文件。但是追加操作从不 损坏文件的现有内容。
[5]	正如你所看到的，原来的行和你以追加方式写入的第二行现在都在 `test.log` 中了。同时注意两行之间并没包含回车符。因为两次写入文件时都没有明确地写入回车符，所以文件中没有包含回车符。你可以用 `"\n"` 写入回车符。因为你没做这项工作，所以你写到文件的所有内容都将显示在同一行上。

进一步阅读

- *Python Tutorial* 讨论了文件的读取和写入，包括如何将一个文件一次一行地读到 list 中。
- eff-bot 讨论了各种各样读取文件方法 的效率和性能。
- Python Knowledge Base 回答了关于文件的常见问题。
- *Python Library Reference* 总结了所有文件对象模块。

6.3. `for` 循环

与其它大多数语言一样，Python 也拥有 `for` 循环。你到现在还未曾看到它们的唯一原因就是，Python 在其它太多的方面表现出色，通常你不需要它们。

其它大多数语言没有像 Python 一样的强大的 list 数据类型，所以你需要亲自做很多事情，指定开始，结束和步长，来定义一定范围的整数或字符或其它可重复的实体。但是在 Python 中，`for` 循环简单地在一个列表上循环，与 list 解析的工作方式相同。

例 6.8. `for` 循环介绍

```
>>> li = ['a', 'b', 'e']
>>> for s in li:
... print s
a
b
e
>>> print "\n".join(li)
a
b
e
```

[1]	`for` 循环的语法同 list 解析相似。`li` 是一个 list，而 `s` 将从第一个元素开始依次接收每个元素的值。
[2]	像 `if` 语句或其它任意缩进块，`for` 循环可以包含任意数目的代码行。
[3]	这就是你以前没看到过 `for` 循环的原因：至今我们都不需要它。太令人吃惊了，当你想要的只是一个 `join` 或是 list 解析时，在其它语言中常常需要使用 `for` 循环。

要做一个 “通常的” (Visual Basic 标准的) 计数 `for` 循环也非常简单。

例 6.9. 简单计数

```
>>> for i in range(5):
... print i
0
1
2
3
4
>>> li = ['a', 'b', 'c', 'd', 'e']
>>> for i in range(len(li)):
... print li[i]
a
b
c
d
e
```

[1]	正如你在 例 3.20 “连续值赋值” 所看到的， `range` 生成一个整数的 list，通过它来控制循环。我知道它看上去有些奇怪，但是它对计数循环偶尔 (我只是说偶尔) 会有用 。
[2]	我们从来没这么用过。这是 Visual Basic 的思维风格。摆脱它吧。正确遍历 list 的方法是前面的例子所展示的。

`for` 循环不仅仅用于简单计数。它们可以遍历任何类型的东西。下面的例子是一个用 `for` 循环遍历 dictionary 的例子。

例 6.10. 遍历 dictionary

```
>>> import os
>>> for k, v in os.environ.items():
... print "%s=%s" % (k, v)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim
[...略...]
>>> print "\n".join(["%s=%s" % (k, v)
... for k, v in os.environ.items()])
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim
[...略...]
```

[1]	`os.environ` 是在你的系统上所定义的环境变量的 dictionary。在 Windows 下，这些变量是可以从 MS-DOS 访问的用户和系统变量。在 UNIX 下，它们是在你的 shell 启动脚本中所 export (输出) 的变量。在 Mac OS 中，没有环境变量的概念，所以这个 dictionary 为空。
[2]	`os.environ.items()` 返回一个 tuple 的 list：`[(_key1_, _value1_), (_key2_, _value2_), ...]`。`for` 循环对这个 list 进行遍历。第一轮，它将 `_key1_` 赋给 `k`，`_value1_` 赋给 `v`，所以 `k = USERPROFILE`，`v = C:\Documents and Settings\mpilgrim`。第二轮，`k` 得到第二个键字 `OS`，`v` 得到相应的值 `Windows_NT`。
[3]	使用多变量赋值和 list 解析，你可以使用单行语句来替换整个 `for` 循环。在实际的编码中是否这样做只是个人风格问题；我喜欢它是因为，将一个 dictionary 映射到一个 list，然后将 list 合并成一个字符串，这一过程显得很清晰。其它的程序员宁愿将其写成一个 `for` 循环。请注意在两种情况下输出是一样的，然而这一版本稍微快一些，因为它只有一条 `print` 语句而不是许多。

现在我们来看看在 第 5 章 介绍的样例程序 `fileinfo.py` 中 `MP3FileInfo` 的 `for` 循环。

例 6.11. `MP3FileInfo` 中的 `for` 循环

```
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}
    .
    .
    .
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
```

[1]	`tagDataMap` 是一个类属性，它定义了我们正在一个 MP3 文件中搜索的标记。标记存储为定长字段，只要我们读出文件最后 128 个字节，那么第 3 到 32 字节总是歌曲的名字，33-62 总是歌手的名字，63-92 为专辑的名字，等等。请注意 `tagDataMap` 是一个 tuple 的 dictionary，每个 tuple 包含两个整数和一个函数引用。
[2]	这个看上去复杂一些，但其实并非如此。这里的 `for` 变量结构与 `items` 所返回的 list 的元素的结构相匹配。记住，`items` 返回一个形如 `(_key_, _value_)` 的 tuple 的 list。list 第一个元素是 `("title", (3, 33, <function stripnulls>))`，所以循环的第一轮，`tag` 为 `"title"`，`start` 为 `3`，`end` 为 `33`，`parseFunc` 为函数 `stripnulls`。
[3]	现在我们已经从一个单个的 MP3 标记中提取出了所有的参数，将标记数据保存起来挺容易。我们从 `start` 到 `end` 对 `tagdata` 进行分片，从而得到这个标记的实际数据，调用 `parseFunc` 对数据进行后续的处理，接着将 `parseFunc` 的返回值作为值赋值给伪字典 `self` 中的键字 `tag`。在遍历完 `tagDataMap` 中所有元素之后，`self` 拥有了所有标记的值，你知道看上去是什么样。

6.4. 使用 `sys .modules`

与其它任何 Python 的东西一样，模块也是对象。只要导入了，总可以用全局 dictionary `sys .modules` 来得到一个模块的引用。

例 6.12. `sys .modules` 介绍

```
>>> import sys
>>> print '\n'.join(sys.modules.keys())
win32api
os.path
os
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
```

[1]	`sys` 模块包含了系统级的信息，像正在运行的 Python 的版本 (`sys`.version` 或 sys
[2]	`sys`.modules`` 是一个字典，它包含了从 Python 开始运行起，被导入的所有模块。键字就是模块名，键值 包含了你在 IDE 中运行的所有程序所导入的所有模块。

下面的例子展示了如何使用 `sys .modules`。

例 6.13. 使用 `sys .modules`

```
>>> import fileinfo
>>> print '\n'.join(sys.modules.keys())
win32api
os.path
os
fileinfo
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
>>> fileinfo
<module 'fileinfo' from 'fileinfo.pyc'>
>>> sys.modules["fileinfo"]
<module 'fileinfo' from 'fileinfo.pyc'>
```

[1]	当导入新的模块，它们加入到 `sys`.modules` 中。这就解释了为什么第二次导入相同的模块时非常的快：Python 已经在 `sys` .modules 中装入和缓冲了，所以第二次导入仅仅对字典做了一个查询。
[2]	一旦给出任何以前导入过的模块名 (以字符串方式)，通过 `sys` .modules` 字典，你可以得到对模块本身的一个引用。

下面的例子将展示通过结合使用 `__module__` 类属性和 `sys` .modules` dictionary 来获取已知类所在的模块。

例 6.14. `__module__` 类属性

```
>>> from fileinfo import MP3FileInfo
>>> MP3FileInfo.__module__
'fileinfo'
>>> sys.modules[MP3FileInfo.__module__]
<module 'fileinfo' from 'fileinfo.pyc'>
```

[1]	每个 Python 类都拥有一个内置的类属性 `__module__`，它定义了这个类的模块的名字。
[2]	将它与 `sys` .modules` 字典复合使用，你可以得到定义了某个类的模块的引用。

现在准备好了，看看在样例程序 第 5 章 `sys` .modules 介绍的 fileinfo.py` 中是如何使用的。这个例子显示它的一部分代码。

例 6.15. `fileinfo.py` 中的 `sys` .modules`

```
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
```

[1]	这是一个有两个参数的函数；`filename` 是必须的，但 `module` 是可选的并且 module 认为 Python 会在每次函数调用时计算这个 `sys` .modules 表达式。实际上，Python 仅会对缺省表达式计算一次，是在模块导入的第一次。正如后面的功能是作为一个函数级别的常量。
[2]	我们会在后面再仔细研究这一行，在我们了解了 `os` 模块之后。那么现在，只要相信
[3]	你已经了解了 `getattr`，它可以通过名字得到一个对象的引用。`hasattr` 是一个补充中，用来检查一个模块是否有一个特别的类 (然而它可以用于任何类和任何属性，就像 class named by `subclass` then return it, otherwise return the base class `FileInfo` (基类 `FileInfo`)”。

进一步阅读

- *Python Tutorial* 讨论了缺省参数到底在什么时候和是如何计算的。
- *Python Library Reference* 提供了 `sys` 模块的文档。

6.5. 与目录共事

`os.path` 模块有几个操作文件和目录的函数。这里，我们看看如何操作路径名和列出一个目录的内容。

例 6.16. 构造路径名

```
>>> import os
>>> os.path.join("c:\\music\\ap\\", "mahadeva.mp3")
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.join("c:\\music\\ap", "mahadeva.mp3")
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.expanduser("~")
'c:\\Documents and Settings\\mpilgrim\\My Documents'
>>> os.path.join(os.path.expanduser("~"), "Python")
'c:\\Documents and Settings\\mpilgrim\\My Documents\\Python'
```

[1]	`os.path` 是一个模块的引用；使用哪一个模块要看你正运行在哪种平台上。就像 `getpass` 通过将 `getpass` 设置为一个与平台相关的函数从而封装了平台之间的不同。`os` 通过设置 `path` 封装不同的相关平台模块。
[2]	`os.path` 的 `join` 函数把一个或多个部分路径名连接成一个路径名。在这个简单的例子中，它只是将字符串进行连接。(请注意在 Windows 下处理路径名是一个麻烦的事，因为反斜线字符必须被转义。)
[3]	在这个几乎没有价值的例子中，在将路径名加到文件名上之前，`join` 将在路径名后添加额外的反斜线。当发现这一点时我高兴极了，因为当用一种新的语言创建我自已的工具包时，`addSlashIfNecessary` 总是我必须要写的那些愚蠢的小函数之一。在 Python 中不要 写这样的愚蠢的小函数，聪明的人已经为你考虑到了。
[4]	`expanduser` 将对使用 `~` 来表示当前用户根目录的路径名进行扩展。在任何平台上，只要用户拥有一个根目录，它就会有效，像 Windows、UNIX 和 Mac OS X，但在 Mac OS 上无效。
[5]	将这些技术组合在一起，你可以容易地为在用户根目录下的目录和文件构造出路径名。

例 6.17. 分割路径名

```
>>> os.path.split("c:\\music\\ap\\mahadeva.mp3")
('c:\\music\\ap', 'mahadeva.mp3')
>>> (filepath, filename) = os.path.split("c:\\music\\ap\\mahadeva.mp3")
>>> filepath
'c:\\music\\ap'
>>> filename
'mahadeva.mp3'
>>> (shortname, extension) = os.path.splitext(filename)
>>> shortname
'mahadeva'
>>> extension
'.mp3'
```

[1]	`split` 函数对一个全路径名进行分割，返回一个包含路径和文件名的 tuple。还记得我说过你可以使用多变量赋值从一个函数返回多个值吗？对，`split` 就是这样一个函数。
[2]	我们将 `split` 函数的返回值赋值给一个两个变量的 tuple。每个变量接收到返回 tuple 相对应的元素值。
[3]	第一个变量，`filepath`，接收到从 `split` 返回 tuple 的第一个元素的值，文件路径。
[4]	第二个变量，`filename`，接收到从 `split` 返回 tuple 的第二个元素的值，文件名。
[5]	`os.path` 也包含了一个 `splitext` 函数，可以用来对文件名进行分割，并且返回一个包含了文件名和文件扩展名的 tuple。我们使用相同的技术来将它们赋值给独立的变量。

例 6.18. 列出目录

```
>>> os.listdir("c:\\music\\_singles\\")
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
>>> dirname = "c:\\"
>>> os.listdir(dirname)
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'cygwin',
'docbook', 'Documents and Settings', 'Incoming', 'Inetpub', 'IO.SYS',
'MSDOS.SYS', 'Music', 'NTDETECT.COM', 'ntldr', 'pagefile.sys',
'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
>>> [f for f in os.listdir(dirname)
... if os.path.isfile(os.path.join(dirname, f))]
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'IO.SYS', 'MSDOS.SYS',
'NTDETECT.COM', 'ntldr', 'pagefile.sys']
>>> [f for f in os.listdir(dirname)
... if os.path.isdir(os.path.join(dirname, f))]
['cygwin', 'docbook', 'Documents and Settings', 'Incoming',
'Inetpub', 'Music', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
```

[1]	`listdir` 函数接收一个路径名，并返回那个目录的内容的 list。
[2]	`listdir` 同时返回文件和文件夹，并不指出哪个是文件，哪个是文件夹。
[3]	你可以使用过滤列表和 `os.path` 模块的 `isfile` 函数，从文件夹中将文件分离出来。 `isfile` 接收一个路径名，如果路径表示一个文件，则返回 1，否则为 0。在这里，我们使用 `os.path`.`join` 来确保得到一个全路径名，但 `isfile` 对部分路径 (相对于当前目录) 也是有效的。你可以使用 `os.getcwd()` 来得到当前目录。
[4]	`os.path` 还有一个 `isdir` 函数，当路径表示一个目录，则返回 1，否则为 0。你可以使用它来得到一个目录下的子目录列表。

例 6.19. 在 `fileinfo.py` 中列出目录

```
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
               for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
```

[1]	`os.listdir(directory)` 返回在 `directory` 中所有文件和文件夹的一个 list。
[2]	使用 `f` 对 list 进行遍历，我们使用 `os.path.normcase(f)` 根据操作系统的缺省值对大小写进行标准化处理。 `normcase` 是一个有用的函数，用于对大小写不敏感操作系统的一个补充。这种操作系统认为 `mahadeva.mp3` 和 `mahadeva.MP3` 是同一个文件名。例如，在 Windows 和 Mac OS 下， `normcase` 将把整个文件名转换为小写字母；而在 UNIX 兼容的系统下，它将返回未作修改的文件名。
[3]	再次用 `f` 对标准化后的 list 进行遍历，我们使用 `os.path.splitext(f)` 将每个文件名分割为名字和扩展名。
[4]	对每个文件，我们查看扩展名是否在我们关心的文件扩展名 list 中 (`fileExtList` ，被传递给 `listDirectory` 函数)。
[5]	对每个我们所关心的文件，我们使用 `os.path.join(directory, f)` 来构造这个文件的全路径名，接着返回这个全路径名的 list。

> 注意
>
> 只要有可能，你就应该使用在 `os` 和 `os.path` 中的函数进行文件、目录和路径的操作。这些模块是对平台相关模块的封装模块，所以像 `os.path.split` 这样的函数可以工作在 UNIX、Windows、Mac OS 和 Python 所支持的任一种平台上。

还有一种获得目录内容的方法。它非常强大，并使用了一些你在命令行上工作时可能已经熟悉的通配符。

例 6.20. 使用 `glob` 列出目录

```
>>> os.listdir("c:\\music\\_singles\\")
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
>>> import glob
>>> glob.glob('c:\\music\\_singles\\*.mp3')
['c:\\music\\_singles\\a_time_long_forgotten_con.mp3',
'c:\\music\\_singles\\hellraiser.mp3',
'c:\\music\\_singles\\kairo.mp3',
'c:\\music\\_singles\\long_way_home1.mp3',
'c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\_singles\\s*.mp3')
['c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\*\\*.mp3')
```

[1]	正如你前面看到的，`os.listdir` 简单地取一个目录路径，返回目录中的所有文件和子目录。
[2]	`glob` 模块，另一方面，接受一个通配符并且返回文件的或目录的完整路径与之匹配。这个通配符是一个目录路径加上“*.mp3”，它将匹配所有的 `.mp3` 文件。注意返回列表的每一个元素已经包含了文件的完整路径。
[3]	如果你要查找指定目录中所有以“s”开头并以“.mp3”结尾的文件，也可以这么做。
[4]	现在考查这种情况：你有一个 `music` 目录，它包含几个子目录，子目录中包含一些 `.mp3` 文件。使用两个通配符，仅仅调用 `glob` 一次就可以立刻获得所有这些文件的一个 list。一个通配符是 `"*.mp3"` (用于匹配 `.mp3` 文件)，另一个通配符是子目录名本身，用于匹配 `c:\music` 中的所有子目录。这看上去很简单，但它蕴含了强大的功能。

进一步阅读

- Python Knowledge Base 回答了关于 `os` 模块的问题。
- *Python Library Reference* 提供了 `os` 模块和 `os.path` 模块的文档。

6.6. 全部放在一起

再一次，所有的多米诺骨牌都放好了。我们已经看过每行代码是如何工作的了。现在往回走一步，看一下放在一起是怎么样的。

例 6.21. `listDirectory`

```
def listDirectory(directory, fileExtList):
   "get list of file info objects for files of particular extensions"
   fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
   fileList = [os.path.join(directory, f)
               for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
   def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
       "get file info class from filename extension"
       subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
       return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
   return [getFileInfoClass(f)(f) for f in fileList]
```

[1]	`listDirectory` 是整个模块主要的有趣之处。它接收一个 dictionary (在我的例子中如 `c:\music\_singles\`) 和一个感兴趣的文件扩展名列表 (如 `['.mp3']`)，接着它返回一个类实例的 list，这些类实例的行为像 dictionary，包含了在目录中每个感兴趣文件的元数据。并且实现起来只用了几行直观的代码。
[2]	正如在前一节我们所看到的，这行代码得到一个全路径名的列表，它的元素是在 `directory` 中有着我们感兴趣的文件后缀 (由 `fileExtList` 所指定的) 的所有文件的路径名。
[3]	老学校出身的 Pascal 程序员可能对嵌套函数感到熟悉，但大部分人，当我告诉他们 Python 支持嵌套函数时，都茫然地看着我。嵌套函数，从字面理解，是定义在函数内的函数。嵌套函数 `getFileInfoClass` 只能在定义它的函数 `listDirectory` 内进行调用。正如任何其它的函数一样，不需要一个接口声明或奇怪的什么东西，只要定义函数，开始编码就行了。
[4]	既然你已经看过 `os` 模块了，这一行应该能理解了。它得到文件的扩展名 (`os.path.splitext(filename)[1]`)，将其转换为大写字母 (`.upper()`)，从圆点处进行分片 (`[1:]`)，使用字符串格式化从其中生成一个类名。所以 `c:\music\ap\mahadeva.mp3` 变成 `.mp3` 再变成 `MP3` 再变成 `MP3FileInfo` 。
[5]	在生成完处理这个文件的处理类的名字之后，我们查阅在这个模块中是否存在这个处理类。如果存在，我们返回这个类，否则我们返回基类 `FileInfo` 。这一点很重要：这个函数返回一个类。不是类的实例，而是类本身。
[6]	对每个属于我们 “感兴趣文件” 列表 (`fileList`)中的文件，我们用文件名 (`f`) 来调用 `getFileInfoClass` 。调用 `getFileInfoClass(f)` 返回一个类；我们并不知道确切是哪一个类，但是我们并不关心。接着我们创建这个类 (不管它是什么) 的一个实例，传入文件名 (又是 `f`) 给 `__init__` 方法。正如我们在本章的前面所看到的， `FileInfo` 的 `__init__` 方法设置了 `self["name"]` ，它将引发 `__setitem__` 的调用，而 `__setitem__` 在子类 (`MP3FileInfo`) 中被覆盖掉了，用来适当地对文件进行分析，取出文件的元数据。我们对所有感兴趣的文件进行处理，返回结果实例的一个 list。

请注意 `listDirectory` 完全是通用的。它事先不知道将得到哪种类型的文件，也不知道哪些定义好的类能够处理这些文件。它检查目录中要进行处理的文件，然后反观本身模块，了解定义了什么特别的处理类 (像 `MP3FileInfo`)。你可以对这个程序进行扩充，对其它类型的文件进行处理，只要用适合的名字定义类： `HTMLFileInfo` 用于 HTML 文件， `DOCFileInfo` 用于 Word `.doc` 文件，等等。不需要改动函数本身， `listDirectory` 将会对它们都进行处理，将工作交给适当的类，接着收集结果。

6.7. 小结

在 第 5 章 介绍的 `fileinfo.py` 程序现在应该完全理解了。

```
"""Framework for getting filetype-specific metadata.
Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key-value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])
Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...
Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict
def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename
class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}
    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass
    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
               for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]
if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]):
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

在研究下一章之前，确保你可以无困难地完成下面的事情：

- 使用 `try...except` 来捕捉异常
- 使用 `try...finally` 来保护额外的资源
- 读取文件
- 在一个 `for` 循环中一次赋多个值
- 使用 `os` 模块来满足你的跨平台文件操作的需要
- 通过将类看成对象并传入参数，动态地实例化未知类型的类

第 7 章 正则表达式

正则表达式是搜索、替换和解析复杂字符模式的一种强大而标准的方法。如果你曾经在其他语言 (如 Perl) 中使用过它，由于它们的语法非常相似，你仅仅阅读一下 `re` 模块的摘要，大致了解其中可用的函数和参数就可以了。

7.1. 概览

字符串也有很多方法，可以进行搜索 (`index` 、 `find` 和 `count`)、替换 (`replace`) 和解析 (`split`)，但它们仅限于处理最简单的情况。搜索方法查找单个和固定编码的子串，并且它们总是大小写敏感的。对一个字符串 `s` ，如果要进行大小写不敏感的搜索，则你必须调用 `s.lower()` 或 `s.upper()` 将 `s` 转换成全小写或者全大写，然后确保搜索串有着相匹配的大小写。 `replace` 和 `split` 方法有着类似的限制。

如果你要解决的问题利用字符串函数能够完成，你应该使用它们。它们快速、简单且容易阅读，而快速、简单、可读性强的代码可以说出很多好处。但是，如果你发现你使用了许多不同的字符串函数和 `if` 语句来处理一个特殊情况，或者你组合使用了 `split` 、 `join` 等函数而导致用一种奇怪的甚至读不下去的方式理解列表，此时，你也许需要转到正则表达式了。

尽管正则表达式语法较之普通代码相对麻烦一些，但是却可以得到更可读的结果，与用一长串字符串函数的解决方案相比要好很多。在正则表达式内部有多种方法嵌入注释，从而使之具有自文档化 (self-documenting) 的能力。

7.2. 个案研究：街道地址

这一系列的例子是由我几年前日常工作中的现实问题启发而来的，当时我需要从一个老化系统中导出街道地址，在将它们导入新的系统之前，进行清理和标准化。(看，我不是只将这些东西堆到一起，它有实际的用处。)这个例子展示我如何处理这个问题。

例 7.1. 在字符串的结尾匹配

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.')
'100 NORTH BROAD RD.'
>>> import re
>>> re.sub('ROAD$', 'RD.', s)
'100 NORTH BROAD RD.'
```

[1]	我的目标是将街道地址标准化， `'ROAD'` 通常被略写为 `'RD.'` 。乍看起来，我以为这个太简单了，只用字符串的方法 `replace` 就可以了。毕竟，所有的数据都已经是大写的了，因此大小写不匹配将不是问题。并且，要搜索的串 `'ROAD'` 是一个常量，在这个迷惑的简单例子中， `s.replace` 的确能够胜任。
[2]	不幸的是，生活充满了特例，并且我很快就意识到这个问题。比如： `'ROAD'` 在地址中出现两次，一次是作为街道名称 `'BROAD'` 的一部分，一次是作为 `'ROAD'` 本身。 `replace` 方法遇到这两处的 `'ROAD'` 并没有区别，因此都进行了替换，而我发现地址被破坏掉了。
[3]	为了解决在地址中出现多次 `'ROAD'` 子串的问题，有可能采用类似这样的方法：只在地址的最后四个字符中搜索替换 `'ROAD'` (s[-4:])，忽略字符串的其他部分 (s[:-4])。但是，你可能发现这已经变得不方便了。例如，该模式依赖于你要替换的字符串的长度了 (如果你要把 `'STREET'` 替换为 `'ST.'` ，你需要利用 `s[:-6]` 和 `s[-6:].replace(...)`)。你愿意在六月个期间回来调试它们么？我本人是不愿意的。
[4]	是时候转到正则表达式了。在 Python 中，所有和正则表达式相关的功能都包含在 `re` 模块中。
[5]	来看第一个参数： `'ROAD$'` 。这个正则表达式非常简单，只有当 `'ROAD'` 出现在一个字符串的尾部时才会匹配。字符 `$` 表示“字符串的末尾”(还有一个对应的字符，尖号 `^` ，表示“字符串的开始”)。
[6]	利用 `re.sub` 函数，对字符串 `s` 进行搜索，满足正则表达式 `'ROAD$'` 的用 `'RD.'` 替换。这样将匹配字符串 `s` 末尾的 `'ROAD'` ，而不会匹配属于单词 `'ROAD'` 一部分的 `'ROAD'` ，这是因为它是出现在 `s` 的中间。

继续我的清理地址的故事。很快我发现，在上面的例子中，仅仅匹配地址末尾的 `'ROAD'` 不是很好，因为不是所有的地址都包括表示街道的单词 (`'ROAD'`)；有一些直接以街道名结尾。大部分情况下，不会遇到这种情况，但是，如果街道名称为 `'BROAD'` ，那么正则表达式将会匹配 `'BROAD'` 的一部分为 `'ROAD'` ，而这并不是我想要的。

例 7.2. 匹配整个单词

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s)
'100 BROAD RD. APT 3'
```

[1]	我真正想要做的是，当 `'ROAD'` 出现在字符串的末尾，并且是作为一个独立的单词时，而不是一些长单词的一部分，才对他进行匹配。为了在正则表达式中表达这个意思，你利用 `\b` ，它的含义是“单词的边界必须在这里”。在 Python 中，由于字符 `'\'` 在一个字符串中必须转义，这会变得非常麻烦。有时候，这类问题被称为“反斜线灾难”，这也是 Perl 中正则表达式比 Python 的正则表达式要相对容易的原因之一。另一方面，Perl 也混淆了正则表达式和其他语法，因此，如果你发现一个 bug，很难弄清楚究竟是一个语法错误，还是一个正则表达式错误。
[2]	为了避免反斜线灾难，你可以利用所谓的“原始字符串”，只要为字符串添加一个前缀 `r` 就可以了。这将告诉 Python，字符串中的所有字符都不转义； `'\t'` 是一个制表符，而 `r'\t'` 是一个真正的反斜线字符 `'\'` ，紧跟着一个字母 `'t'` 。我推荐只要处理正则表达式，就使用原始字符串；否则，事情会很快变得混乱 (并且正则表达式自己也会很快被自己搞乱了)。
[3]	(一声叹息) 很不幸，我很快发现更多的与我的逻辑相矛盾的例子。在这个例子中，街道地址包含有作为整个单词的 `'ROAD'` ，但是它不是在末尾，因为地址在街道命名后会有一个房间号。由于 `'ROAD'` 不是在每一个字符串的末尾，没有匹配上，因此调用 `re.sub` 没有替换任何东西，你获得的只是初始字符串，这也不是我们想要的。
[4]	为了解决这个问题，我去掉了 `$` 字符，加上另一个 `\b` 。现在，正则表达式“匹配字符串中作为整个单词出现的 `'ROAD'` ”了，不论是在末尾、开始还是中间。

7.3. 个案研究：罗马字母

你可能经常看到罗马数字，即使你没有意识到它们。你可能曾经在老电影或者电视中看到它们 (“版权所有 `MCMXLVI` ” 而不是 “版权所有 `1946` ”)，或者在某图书馆或某大学的贡献墙上看到它们 (“成立于 `MDCCCLXXXVIII` ”而不是“成立于 `1888` ”)。你也可能在某些文献的大纲或者目录上看到它们。这是一个表示数字的系统，它实际上能够追溯到远古的罗马帝国 (因此而得名)。

在罗马数字中，利用7个不同字母进行重复或者组合来表达各式各样的数字。

- `I` = `1`
- `V` = `5`
- `X` = `10`
- `L` = `50`
- `C` = `100`
- `D` = `500`
- `M` = `1000`

下面是关于构造罗马数字的一些通用的规则的介绍：

- 字符是叠加的。 `I` 表示 `1`， `II` 表示 `2`， 而 `III` 表示 `3` 。 `VI` 表示 `6` (字面上为逐字符相加，“ `5` 加 `1` ”)， `VII` 表示 `7` ， `VIII` 表示 `8` 。
- 含十字符 (`I` 、 `X` 、 `C` 和 `M`) 至多可以重复三次。对于 `4` ，你则需要利用下一个最大的含五字符进行减操作得到：你不能把 `4` 表示成 `IIII` ，而应表示为 `IV` (“比 `5` 小 `1` ”)。数字 `40` 写成 `XL` (比 `50` 小 `10`)， `41` 写成 `XLI` ， `42` 写成 `XLII` ， `43` 写成 `XLIII` ， 而 `44` 写成 `XLIV` (比 `50` 小 `10` ， 然后比 `5` 小 `1`)。
- 类似地，对于数字 `9` ，你必须利用下一个含十字符进行减操作得到： `8` 表示为 `VIII` ， 而 `9` 则表示为 `IX` (比 `10` 小 `1`)，而不是 `VIIII` (因为字符 `I` 不能连续重复四次)。数字 `90` 表示为 `XC` ， `900` 表示为 `CM` 。
- 含五字符不能重复。数字 `10` 常表示为 `X` ，而从来不用 `VV` 来表示。数字 `100` 常表示为 `C` ，也从来不表示为 `LL` 。
- 罗马数字一般从高位到低位书写，从左到右阅读，因此不同顺序的字符意义大不相同。 `DC` 表示 `600` ；而 `CD` 是一个完全不同的数字 (为 `400` ，也就是比 `500` 小 `100`)。 `CI` 表示 `101` ；而 `IC` 甚至不是一个合法的罗马字母 (因为你不能直接从数字 `100` 减去 `1` ；这需要写成 `XCIX` ，意思是比 `100` 小 `10` ，然后加上数字 `9` ，也就是比 `10` 小 `1` 的数字)。

7.3.1. 校验千位数

怎样校验任意一个字符串是否为一个有效的罗马数字呢？我们每次只看一位数字，由于罗马数字一般是从高位到低位书写。我们从高位开始：千位。对于大于或等于 1000 的数字，千位由一系列的字符 `M` 表示。

例 7.3. 校验千位数

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')
<SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')
<SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')
<SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM')
>>> re.search(pattern, '')
<SRE_Match object at 0106F4A8>
```

[1]	这个模式有三部分：`^` 表示仅在一个字符串的开始匹配其后的字符串内容。如果没有这个字符，这个模式将匹配出现在字符串任意位置上的 `M`，而这并不是你想要的。你想确认的是：字符串中是否出现字符 `M`，如果出现，则必须是在字符串的开始。`M?` 可选地匹配单个字符 `M`，由于它最多可重复出现三次，你可以在一行中匹配 0 次到 3 次字符 `M`。`$` 字符限制模式只能够在一个字符串的结尾匹配。当和模式开头的字符 `^` 结合使用时，这意味着模式必须匹配整个串，并且在在字符 `M` 的前后都不能够出现其他的任意字符。
[2]	`re` 模块的关键是一个 `search` 函数，该函数有两个参数，一个是正则表达式 (`pattern`)，一个是字符串 (`'M'`)，函数试图匹配正则表达式。如果发现一个匹配，`search` 函数返回一个拥有多种方法可以描述这个匹配的对象，如果没有发现匹配，`search` 函数返回一个 `None`，一个 Python 空值 (null value)。你此刻关注的唯一事情，就是模式是否匹配上，于是我们利用 `search` 函数的返回值了解这个事实。字符串 `'M'` 匹配上这个正则表达式，因为第一个可选的 `M` 匹配上，而第二个和第三个 `M` 被忽略掉了。
[3]	`'MM'` 能匹配上是因为第一和第二个可选的 `M` 匹配上，而忽略掉第三个 `M`。
[4]	`'MMM'` 能匹配上因为三个 `M` 都匹配上了。
[5]	`'MMMM'` 没有匹配上。因为所有的三个 `M` 都匹配完了，但是正则表达式还有字符串尾部的限制 (由于字符 `$`)，而字符串又没有结束 (因为还有第四个 `M` 字符)，因此 `search` 函数返回一个 `None`。
[6]	有趣的是，一个空字符串也能够匹配这个正则表达式，因为所有的字符 `M` 都是可选的。

7.3.2. 校验百位数

与千位数相比，百位数识别起来要困难得多，这是因为有多种相互独立的表达方式都可以表达百位数，而具体用那种方式表达和具体的数值有关。

- `100` = `C`
- `200` = `CC`
- `300` = `CCC`
- `400` = `CD`
- `500` = `D`
- `600` = `DC`
- `700` = `DCC`
- `800` = `DCCC`
- `900` = `CM`

因此有四种可能的模式：

- `CM`
- `CD`
- 零到三次出现 `C` 字符 (出现零次表示百位数为 0)
- `D`，后面跟零个到三个 `C` 字符

后面两个模式可以结合到一起：

- 一个可选的字符 `D`，加上零到 3 个 `C` 字符。

这个例子显示如何有效地识别罗马数字的百位数。

例 7.4. 检验百位数

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$'
>>> re.search(pattern, 'MCM')
<SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')
<SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')
<SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')
>>> re.search(pattern, '')
<SRE_Match object at 01071D98>
```

[1]	这个模式的首部和上一个模式相同，检查字符串的开始 (`^`)，接着匹配千位数 (`M?M?M?`)，然后才是这个模式的新内容。在括号内，定义了包含有三个互相独立的模式集合，由垂直线隔开：`CM` 、 `CD` 和 `D?C?C?C?` (`D` 是可选字符，接着是 0 到 3 个可选的 `C` 字符)。正则表达式解析器依次检查这些模式 (从左到右)，如果匹配上第一个模式，则忽略剩下的模式。
[2]	`'MCM'` 匹配上，因为第一个 `M` 字符匹配，第二和第三个 `M` 字符被忽略掉，而 `CM` 匹配上 (因此 `CD` 和 `D?C?C?C?` 两个模式不再考虑)。 `MCM` 表示罗马数字 `1900` 。
[3]	`'MD'` 匹配上，因为第一个字符 `M` 匹配上，第二第三个 `M` 字符忽略，而模式 `D?C?C?C?` 匹配上 `D` (模式中的三个可选的字符 `C` 都被忽略掉了)。 `MD` 表示罗马数字 `1500` 。
[4]	`'MMMCCC'` 匹配上，因为三个 `M` 字符都匹配上，而模式 `D?C?C?C?` 匹配上 `CCC` (字符 `D` 是可选的，此处忽略)。 `MMMCCC` 表示罗马数字 `3300` 。
[5]	`'MCMC'` 没有匹配上。第一个 `M` 字符匹配上，第二第三个 `M` 字符忽略，接着是 `CM` 匹配上，但是接着是 `$` 字符没有匹配，因为字符串还没有结束 (你仍然还有一个没有匹配的 `C` 字符)。 `C` 字符也不 匹配模式 `D?C?C?C?` 的一部分，因为与之相互独立的模式 `CM` 已经匹配上。
[6]	有趣的是，一个空字符串也可以匹配这个模式，因为所有的 `M` 字符都是可选的，它们都被忽略，并且一个空字符串可以匹配 `D?C?C?C?` 模式，此处所有的字符也都是可选的，并且都被忽略。

哎呀！看看正则表达式能够多快变得难以理解？你仅仅表示了罗马数字的千位和百位上的数字。如果你根据类似的方法，十位数和各位数就非常简单了，因为是完全相同的模式。让我们来看表达这个模式的另一种方式吧。

7.4. 使用 `{n,m}` 语法

- 7.4.1. 校验十位数和个位数

在前面的章节，你处理了相同字符可以重复三次的情况。在正则表达式中，有另外一个方式来表达这种情况，并且能提高代码的可读性。首先看看我们在前面的例子中使用的方法。

例 7.5. 老方法：每一个字符都是可选的

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')
<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM')
<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM')
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM')
>>>
```

[1]	这个模式匹配串的开始，接着是第一个可选的字符 `M`，第二第三个 `M` 字符则被忽略 (这是可行的，因为它们都是可选的)，最后是字符串的结尾。
[2]	这个模式匹配串的开始，接着是第一和第二个可选字符 `M`，而第三个 `M` 字符被忽略 (这是可行的，因为它们都是可选的)，最后匹配字符串的结尾。
[3]	这个模式匹配字符串的开始，接着匹配所有的三个可选字符 `M`，最后匹配字符串的结尾。
[4]	这个模式匹配字符串的开始，接着匹配所有的三个可选字符 `M`，但是不能够匹配字符串的结尾 (因为还有一个未匹配的字符 `M`)，因此不能够匹配而返回一个 `None`。

例 7.6. 一个新的方法：从 `n` 到 `m`

```
>>> pattern = '^M{0,3}$'
>>> re.search(pattern, 'M')
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM')
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM')
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM')
>>>
```

[1]	这个模式意思是说：“匹配字符串的开始，接着匹配 0 到 3 个 `M` 字符，然后匹配字符串的结尾。”这里的 0 和 3 可以改成其它任何数字；如果你想要匹配至少 1 次，至多 3 次字符 `M`，则可以写成 `M{1,3}`。
[2]	这个模式匹配字符串的开始，接着匹配三个可选 `M` 字符中的一个，最后是字符串的结尾。
[3]	这个模式匹配字符串的开始，接着匹配三个可选 `M` 字符中的两个，最后是字符串的结尾。
[4]	这个模式匹配字符串的开始，接着匹配三个可选 `M` 字符中的三个，最后是字符串的结尾。
[5]	这个模式匹配字符串的开始，接着匹配三个可选 `M` 字符中的三个，但是没有匹配上 字符串的结尾。正则表达式在字符串结尾之前最多只允许匹配三次 `M` 字符，但是实际上有四个 `M` 字符，因此模式没有匹配上这个字符串，返回一个 `None`。

> 注意
> 没有一个轻松的方法来确定两个正则表达式是否等价。你能采用的最好的办法就是列出很多的测试样例，确定这两个正则表达式对所有的相关输入都有相同的输出。在本书后面的章节，将更多地讨论如何编写测试样例。

7.4.1. 校验十位数和个位数

现在我们来扩展一下关于罗马数字的正则表达式，以匹配十位数和个位数，下面的例子展示十位数的校验方法。

例 7.7. 校验十位数

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX')
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX')
>>>
```

[1]	这个模式匹配字符串的开始，接着是第一个可选字符 `M`，接着是 `CM`，接着 `XL`，接着是字符串的结尾。请记住，`(A|B|C)` 这个语法的含义是“精确匹配 A、B 或者 C 其中的一个”。此处匹配了 `XL`，因此不再匹配 `XC` 和 `L?X?X?X?`，接着就匹配到字符串的结尾。`MCML` 表示罗马数字 `1940`。
[2]	这个模式匹配字符串的开始，接着是第一个可选字符 `M`，接着是 `CM`，接着 `L?X?X?X?`。在模式 `L?X?X?X?` 中，它匹配 `L` 字符并且跳过所有可选的 `X` 字符，接着匹配字符串的结尾。`MCML` 表示罗马数字 `1950`。
[3]	这个模式匹配字符串的开始，接着是第一个可选字符 `M`，接着是 `CM`，接着是可选的 `L` 字符和可选的第一个 `X` 字符，并且跳过第二第三个可选的 `X` 字符，接着是字符串的结尾。`MCMLX` 表示罗马数字 `1960`。
[4]	这个模式匹配字符串的开始，接着是第一个可选字符 `M`，接着是 `CM`，接着是可选的 `L` 字符和所有的三个可选的 `X` 字符，接着匹配字符串的结尾。`MCMLXXX` 表示罗马数字 `1980`。
[5]	这个模式匹配字符串的开始，接着是第一个可选字符 `M`，接着是 `CM`，接着是可选的 `L` 字符和所有的三个可选的 `X` 字符，接着就未能匹配 字符串的结尾ie，因为还有一个未匹配的 `X` 字符。所以整个模式匹配失败并返回一个 `None`. `MCMLXXXX` 不是一个有效的罗马数字。

对于个位数的正则表达式有类似的表达方式，我将省略细节，直接展示结果。

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

用另一种 `{n,m}` 语法表达这个正则表达式会如何呢？这个例子展示新的语法。

例 7.8. 用 `{n,m}` 语法确认罗马数字

```
>>> pattern = '^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV')
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI')
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCCLXXXVIII')
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I')
<_sre.SRE_Match object at 0x008EEB48>
```

[1]	这个模式匹配字符串的开始，接着匹配三个可选的 `M` 字符的一个，接着匹配 `D?C{0,3}`，此处，仅仅匹配可选的字符 `D` 和 0 个可选字符 `C`。继续向前匹配，匹配 `L?X{0,3}`，此处，匹配可选的 `L` 字符和 0 个可选字符 `X`，接着匹配 `V?I{0,3}`，此处，匹配可选的 V 和 0 个可选字符 `I`，最后匹配字符串的结尾。`MDLV` 表示罗马数字 `1555`。
[2]	这个模式匹配字符串的开始，接着是三个可选的 `M` 字符的两个，接着匹配 `D?C{0,3}`，此处为一个字符 `D` 和三个可选 `C` 字符中的一个，接着匹配 `L?X{0,3}`，此处为一个 `L` 字符和三个可选 `X` 字符中的一个，接着匹配 `V?I{0,3}`，此处为一个字符 `V` 和三个可选 `I` 字符中的一个，接着匹配字符串的结尾。`MMDCLXVI` 表示罗马数字 `2666`。
[3]	这个模式匹配字符串的开始，接着是三个可选的 `M` 字符的所有字符，接着匹配 `D?C{0,3}`，此处为一个字符 `D` 和三个可选 `C` 字符中所有字符，接着匹配 `L?X{0,3}`，此处为一个 `L` 字符和三个可选 `X` 字符中所有字符，接着匹配 `V?I{0,3}`，此处为一个字符 `V` 和三个可选 `I` 字符中所有字符，接着匹配字符串的结尾。`MMMDCCCLXXXVIII` 表示罗马数字 `3888`，这个数字是不用扩展语法可以写出的最大的罗马数字。
[4]	仔细看哪！(我像一个魔术师一样，“看仔细喽，孩子们，我将要从我的帽子中拽出一只兔子来啦！”) 这个模式匹配字符串的开始，接着匹配 3 个可选 `M` 字符的 0 个，接着匹配 `D?C{0,3}`，此处，跳过可选字符 `D` 并匹配三个可选 `C` 字符的 0 个，接着匹配 `L?X{0,3}`，此处，跳过可选字符 `L` 并匹配三个可选 `X` 字符的 0 个，接着匹配 `V?I{0,3}`，此处跳过可选字符 `V` 并匹配三个可选 `I` 字符的一个，最后匹配字符串的结尾。哇赛！

如果你在第一遍就跟上并理解了所讲的这些，那么你做的比我还要好。现在，你可以尝试着理解别人大规模程序里关键函数中的正则表达式了。或者想象着几个月后回头理解你自己的正则表达式。我曾经做过这样的事情，但是它并不是那么有趣。

在下一节里，你将会研究另外一种正则表达式语法，它可以使你的表达式具有更好的可维持性。

7.5. 松散正则表达式

迄今为止，你只是处理过被我称之为“紧凑”类型的正则表达式。正如你曾看到的，它们难以阅读，即使你清楚正则表达式的含义，你也不能保证六个月以后你还能理解它。你真正所需的就是利用内联文档 (inline documentation)。

Python 允许用户利用所谓的松散正则表达式 来完成这个任务。一个松散正则表达式和一个紧凑正则表达式主要区别表现在两个方面：

- 忽略空白符。空格符，制表符，回车符不匹配它们自身，它们根本不参与匹配。(如果你想在松散正则表达式中匹配一个空格符，你必须在它前面添加一个反斜线符号对它进行转义。)
- 忽略注释。在松散正则表达式中的注释和在普通 Python 代码中的一样：开始于一个 `#` 符号，结束于行尾。这种情况下，采用在一个多行字符串中注释，而不是在源代码中注释，它们以相同的方式工作。

用一个例子可以解释得更清楚。让我们重新来看前面的紧凑正则表达式，利用松散正则表达式重新表达。下面的例子显示实现方法。

例 7.9. 带有内联注释 (Inline Comments) 的正则表达式

```
>>> pattern = """
    ^                   # beginning of string
    M{0,3}              # thousands - 0 to 3 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    """
>>> re.search(pattern, 'M', re.VERBOSE)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCCLXXXVIII', re.VERBOSE)
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'M')
```

[1]	当使用松散正则表达式时，最重要的一件事情就是：必须传递一个额外的参数 `re.VERBOSE`，该参数是定义在 `re` 模块中的一个常量，标志着待匹配的正则表达式是一个松散正则表达式。正如你看到的，这个模式中，有很多空格 (所有的空格都被忽略)，和几个注释 (所有的注释也被忽略)。如果忽略所有的空格和注释，它就和前面章节里的正则表达式完全相同，但是具有更好的可读性。
[2]	这个模式匹配字符串的开始，接着匹配三个可选 `M` 字符中的一个，接着匹配 `CM`，接着是字符 `L` 和三个可选 `X` 字符的所有字符，接着是 `IX`，然后是字符串的结尾。
[3]	这个模式匹配字符串的开始，接着是三个可选的 `M` 字符的所有字符，接着匹配 `D?C{0,3}`，此处为一个字符 `D` 和三个可选 `C` 字符中所有字符，接着匹配 `L?X{0,3}`，此处为一个 `L` 字符和三个可选 `X` 字符中所有字符，接着匹配 `V?I{0,3}`，此处为一个字符 `V` 和三个可选 `I` 字符中所有字符，接着匹配字符串的结尾。
[4]	这个没有匹配。为什么呢？因为没有 `re.VERBOSE` 标记，所以 `re.search` 函数把模式作为一个紧凑正则表达式进行匹配。Python 不能自动检测一个正则表达式是为松散类型还是紧凑类型。Python 默认每一个正则表达式都是紧凑类型的，除非你显式地标明一个正则表达式为松散类型。

7.6. 个案研究：解析电话号码

迄今为止，你主要是匹配整个模式，不论是匹配上，还是没有匹配上。但是正则表达式还有比这更为强大的功能。当一个模式确实 匹配上时，你可以获取模式中特定的片断，你可以发现具体匹配的位置。

这个例子来源于我遇到的另一个现实世界的问题，也是在以前的工作中遇到的。问题是：解析一个美国电话号码。客户要能 (在一个单一的区域中) 输入任何数字，然后存储区号、干线号、电话号和一个可选的独立的分机号到公司数据库里。为此，我通过网络找了很多正则表达式的例子，但是没有一个能够完全满足我的要求。

这里列举了我必须能够接受的电话号码：

- `800-555-1212`
- `800 555 1212`
- `800.555.1212`
- `(800) 555-1212`
- `1-800-555-1212`
- `800-555-1212-1234`
- `800-555-1212x1234`
- `800-555-1212 ext. 1234`
- `work 1-(800) 555.1212 #1234`

格式可真够多的！我需要知道区号是 `800`，干线号是 `555`，电话号的其他数字为 `1212`。对于那些有分机号的，我需要知道分机号为 `1234`。

让我们完成电话号码解析这个工作，这个例子展示第一步。

例 7.10. 发现数字

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$')
>>> phonePattern.search('800-555-1212').groups()
('800', '555', '1212')
>>> phonePattern.search('800-555-1212-1234')
>>>
```

[1]	我们通常从左到右阅读正则表达式。这个正则表达式匹配字符串的开始，接着匹配 `(\d{3})` 。 `\d{3}` 是什么呢？好吧， `{3}` 的含义是“精确匹配三个数字”；这是曾在前面见到过的 `{n,m}` 语法的一种变形。 `\d` 的含义是 “任何一个数字” (`0` 到 `9`)。把它们放大括号中意味着要“精确匹配三个数字位，接着把它们作为一个组保存下来，以便后面的调用”。接着匹配一个连字符，接着是另外一个精确匹配三个数字位的组，接着另外一个连字符，接着另外一个精确匹配四个数字为的组，接着匹配字符串的结尾。
[2]	为了访问正则表达式解析过程中记忆下来的多个组，我们使用 `search` 函数返回对象的 `groups()` 函数。这个函数将返回一个元组，元组中的元素就是正则表达式中定义的组。在这个例子中，定义了三个组，第一个组有三个数字位，第二个组有三个数字位，第三个组有四个数字位。
[3]	这个正则表达式不是最终的答案，因为它不能处理在电话号码结尾有分机号的情况，为此，我们需要扩展这个正则表达式。

例 7.11. 发现分机号

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$')
>>> phonePattern.search('800-555-1212-1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')
>>>
>>> phonePattern.search('800-555-1212')
>>>
```

[1]	这个正则表达式和上一个几乎相同，正像前面的那样，匹配字符串的开始，接着匹配一个有三个数字位的组并记忆下来，接着是一个连字符，接着是一个有三个数字位的组并记忆下来，接着是一个连字符，接着是一个有四个数字位的组并记忆下来。不同的地方是你接着又匹配了另一个连字符，然后是一个有一个或者多个数字位的组并记忆下来，最后是字符串的结尾。
[2]	函数 `groups()` 现在返回一个有四个元素的元组，由于正则表达式中定义了四个记忆的组。
[3]	不幸的是，这个正则表达式也不是最终的答案，因为它假设电话号码的不同部分是由连字符分割的。如果一个电话号码是由空格符、逗号或者点号分割呢？你需要一个更一般的解决方案来匹配几种不同的分割类型。
[4]	啊呀！这个正则表达式不仅不能解决你想要的任何问题，反而性能更弱了，因为现在你甚至不能解析一个没有分机号的电话号码了。这根本不是你想要的，如果有分机号，你要知道分机号是什么，如果没有分机号，你仍然想要知道主电话号码的其他部分是什么。

下一个例子展示正则表达式处理一个电话号码内部，采用不同分隔符的情况。

例 7.12. 处理不同分隔符

```
>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$')
>>> phonePattern.search('800 555 1212 1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234')
>>>
>>> phonePattern.search('800-555-1212')
>>>
```

[1]	当心啦！你首先匹配字符串的开始，接着是一个三个数字位的组，接着是 `\D+`，这是个什么东西？好吧，`\D` 匹配任意字符，除了 数字位，`+` 表示“1 个或者多个”，因此 `\D+` 匹配一个或者多个不是数字位的字符。这就是你替换连字符为了匹配不同分隔符所用的方法。
[2]	使用 `\D+` 代替 `-` 意味着现在你可以匹配中间是空格符分割的电话号码了。
[3]	当然，用连字符分割的电话号码也能够被识别。
[4]	不幸的是，这个正则表达式仍然不是最终答案，因为它假设电话号码一定有分隔符。如果电话号码中间没有空格符或者连字符的情况会怎样哪？
[4]	我的天！这个正则表达式也没有达到我们对于分机号识别的要求。现在你共有两个问题，但是你可以利用相同的技术来解决它们。

下一个例子展示正则表达式处理没有 分隔符的电话号码的情况。

例 7.13. 处理没有分隔符的数字

```
>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
>>> phonePattern.search('80055512121234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234')
>>>
```

[1]	和上一步相比，你所做的唯一变化就是把所有的 `+` 变成 `*`。在电话号码的不同部分之间不再匹配 `\D+`，而是匹配 `\D*` 了。还记得 `+` 的含义是“1 或者多个”吗? 好的，`*` 的含义是“0 或者多个”。因此，现在你应该能够解析没有分隔符的电话号码了。
[2]	你瞧，它真的可以胜任。为什么？首先匹配字符串的开始，接着是一个有三个数字位 (`800`) 的组，接着是 0 个非数字字符，接着是一个有三个数字位 (`555`) 的组，接着是 0 个非数字字符，接着是一个有四个数字位 (`1212`) 的组，接着是 0 个非数字字符，接着是一个有任意数字位 (`1234`) 的组，最后是字符串的结尾。
[3]	对于其他的变化也能够匹配：比如点号分隔符，在分机号前面既有空格符又有 `x` 符号的情况也能够匹配。
[4]	最后，你已经解决了长期存在的一个问题：现在分机号是可选的了。如果没有发现分机号， `groups()` 函数仍然返回一个有四个元素的元组，但是第四个元素只是一个空字符串。
[5]	我不喜欢做一个坏消息的传递人，此时你还没有完全结束这个问题。还有什么问题呢？当在区号前面还有一个额外的字符时，而正则表达式假设区号是一个字符串的开始，因此不能匹配。这个不是问题，你可以利用相同的技术“0或者多个非数字字符”来跳过区号前面的字符。

下一个例子展示如何解决电话号码前面有其他字符的情况。

例 7.14. 处理开始字符

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
>>> phonePattern.search('(800)5551212 ext. 1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234')
>>>
```

[1]	这个正则表达式和前面的几乎相同，但它在第一个记忆组 (区号) 前面匹配 `\D*`，0 或者多个非数字字符。注意，此处你没有记忆这些非数字字符 (它们没有被括号括起来)。如果你发现它们，只是跳过它们，接着只要匹配上就开始记忆区号。
[2]	你可以成功地解析电话号码，即使在区号前面有一个左括号。(在区号后面的右括号也已经被处理，它被看成非数字字符分隔符，由第一个记忆组后面的 `\D*` 匹配。)
[3]	进行仔细的检查，保证你没有破坏前面能够匹配的任何情况。由于首字符是完全可选的，这个模式匹配字符串的开始，接着是 0 个非数字字符，接着是一个有三个数字字符的记忆组 (`800`)，接着是 1 个非数字字符 (连字符)，接着是一个有三个数字字符的记忆组 (`555`)，接着是 1 个非数字字符 (连字符)，接着是一个有四个数字字符的记忆组 (`1212`)，接着是 0 个非数字字符，接着是一个有 0 个数字位的记忆组，最后是字符串的结尾。
[4]	此处是正则表达式让我产生了找一个硬东西挖出自己的眼睛的冲动。为什么这个电话号码没有匹配上？因为在它的区号前面有一个 `1`，但是你认为在区号前面的所有字符都是非数字字符 (`\D*`)。唉！

让我们往回看一下。迄今为止，正则表达式总是从一个字符串的开始匹配。但是现在你看到了，有很多不确定的情况需要你忽略。与其尽力全部匹配它们，还不如全部跳过它们，让我们采用一个不同的方法：根本不显式地匹配字符串的开始。下面的这个例子展示这个方法。

例 7.15. 电话号码，无论何时我都要找到它

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234')
('800', '555', '1212', '1234')
```

[1]	注意，在这个正则表达式的开始少了一个 `^` 字符。你不再匹配字符串的开始了，也就是说，你需要用你的正则表达式匹配整个输入字符串，除此之外没有别的意思了。正则表达式引擎将要努力计算出开始匹配输入字符串的位置，并且从这个位置开始匹配。
[2]	现在你可以成功解析一个电话号码了，无论这个电话号码的首字符是不是数字，无论在电话号码各部分之间有多少任意类型的分隔符。
[3]	仔细检查，这个正则表达式仍然工作的很好。
[4]	还是能够工作。

看看一个正则表达式能够失控得多快？回头看看前面的例子，你还能区别它们么？

当你还能够理解这个最终答案的时候 (这个正则表达式就是最终答案，即使你发现一种它不能处理的情况，我也真的不想知道它了)，在你忘记为什么你这么选择之前，让我们把它写成松散正则表达式的形式。

例 7.16. 解析电话号码 (最终版本)

```
>>> phonePattern = re.compile(r'''
                # don't match beginning of string, number can start anywhere
    (\d{3})     # area code is 3 digits (e.g. '800')
    \D*         # optional separator is any number of non-digits
    (\d{3})     # trunk is 3 digits (e.g. '555')
    \D*         # optional separator
    (\d{4})     # rest of number is 4 digits (e.g. '1212')
    \D*         # optional separator
    (\d*)       # extension is optional and can be any number of digits
    $           # end of string
    ''', re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')
('800', '555', '1212', '')
```

[1]	除了被分成多行，这个正则表达式和最后一步的那个完全相同，因此它能够解析相同的输入一点也不奇怪。
[2]	进行最后的仔细检查。很好，仍然工作。你终于完成了这件任务。

关于正则表达式的进一步阅读

- Regular Expression HOWTO 讲解正则表达式和如何在 Python 中使用正则表达式。
- *Python Library Reference* 概述了 `re` module。

7.7. 小结

这只是正则表达式能够完成工作的很少一部分。换句话说，即使你现在备受打击，相信我，你也不是什么也没见过了。

现在，你应该熟悉下列技巧：

- `^` 匹配字符串的开始。
- `$` 匹配字符串的结尾。
- `\b` 匹配一个单词的边界。
- `\d` 匹配任意数字。
- `\D` 匹配任意非数字字符。
- `x?` 匹配一个可选的 `x` 字符 (换言之，它匹配 1 次或者 0 次 `x` 字符)。
- `x*` 匹配0次或者多次 `x` 字符。
- `x+` 匹配1次或者多次 `x` 字符。
- `x{n,m}` 匹配 `x` 字符，至少 `n` 次，至多 `m` 次。
- `(a|b|c)` 要么匹配 `a`，要么匹配 `b`，要么匹配 `c`。
- `(x)` 一般情况下表示一个记忆组 *(remembered group)*。你可以利用 `re.search` 函数返回对象的 `groups()` 函数获取它的值。

正则表达式非常强大，但是它并不能为每一个问题提供正确的解决方案。你应该学习足够多的知识，以辨别什么时候它们是合适的，什么时候它们会解决你的问题，什么时候它们产生的问题比要解决的问题还要多。

> 一些人，遇到一个问题时就想："我知道，我将使用正则表达式。"现在他有两个问题了。
> --Jamie Zawinski, in comp.emacs.xemacs

第 8 章 HTML 处理

8.1. 概览

我经常在 comp.lang.python 上看到关于如下的问题： “ 怎么才能从我的 HTML 文档中列出所有的 [头|图像|链接] 呢？” “怎么才能 [分析|解释|munge] 我的 HTML 文档的文本，但是又要保留标记呢？” “怎么才能一次给我所有的 HTML 标记 [增加|删除|加引号] 属性呢？” 本章将回答所有这些问题。

下面给出一个完整的，可工作的 Python 程序，它分为两部分。第一部分， `BaseHTMLProcessor.py` 是一个通用工具，它可以通过扫描标记和文本块来帮助您处理 HTML 文件。第二部分， `dialect.py` 是一个例子，演示了如何使用 `BaseHTMLProcessor.py` 来转化 HTML 文档，保留文本但是去掉了标记。阅读文档字符串 (`doc string`) 和注释来了解将要发生事情的概况。大部分内容看上去像巫术，因为任一个这些类的方法是如何调用的不是很清楚。不要紧，所有内容都会按进度被逐步地展示出来。

例 8.1. `BaseHTMLProcessor.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
from sgmllib import SGMLParser
import htmlentitydefs
class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        # extend (called by SGMLParser.__init__)
        self.pieces = []
        SGMLParser.reset(self)
    def unknown_starttag(self, tag, attrs):
        # called for each start tag
        # attrs is a list of (attr, value) tuples
        # e.g. for <pre class="screen">, tag="pre", attrs=[("class", "screen")]
        # Ideally we would like to reconstruct original tag and attributes, but
        # we may end up quoting attribute values that weren't quoted in the source
        # document, or we may change the type of quotes around the attribute value
        # (single to double quotes).
        # Note that improperly embedded non-HTML code (like client-side Javascript)
        # may be parsed incorrectly by the ancestor, causing runtime script errors.
        # All non-HTML code must be enclosed in HTML comment tags (<!-- code -->)
        # to ensure that it will pass through this parser unaltered (in handle_comment).
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
    def unknown_endtag(self, tag):
        # called for each end tag, e.g. for </pre>, tag will be "pre"
        # Reconstruct the original end tag.
        self.pieces.append("</%(tag)s>" % locals())
    def handle_charref(self, ref):
        # called for each character reference, e.g. for " ", ref will be "160"
        # Reconstruct the original character reference.
        self.pieces.append("&#%(ref)s;" % locals())
    def handle_entityref(self, ref):
        # called for each entity reference, e.g. for "&copy;", ref will be "copy"
        # Reconstruct the original entity reference.
        self.pieces.append("&%(ref)s" % locals())
        # standard HTML entities are closed with a semicolon; other entities are not
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")
    def handle_data(self, text):
        # called for each block of plain text, i.e. outside of any tag and
        # not containing any character or entity references
        # Store the original text verbatim.
        self.pieces.append(text)
    def handle_comment(self, text):
        # called for each HTML comment, e.g. <!-- insert Javascript code here -->
        # Reconstruct the original comment.
        # It is especially important that the source document enclose client-side
        # code (like Javascript) within comments so it can pass through this
        # processor undisturbed; see comments in unknown_starttag for details.
        self.pieces.append("<!--%(text)s-->" % locals())
    def handle_pi(self, text):
        # called for each processing instruction, e.g. <?instruction>
        # Reconstruct original processing instruction.
        self.pieces.append("<?%(text)s>" % locals())
    def handle_decl(self, text):
        # called for the DOCTYPE, if present, e.g.
        # <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        #     "http://www.w3.org/TR/html4/loose.dtd">
        # Reconstruct original DOCTYPE
        self.pieces.append("<!%(text)s>" % locals())
    def output(self):
        """Return processed HTML as a single string"""
        return "".join(self.pieces)
```

例 8.2. `dialect.py`

```
import re
```

```
from BaseHTMLProcessor import BaseHTMLProcessor
class Dialectizer(BaseHTMLProcessor):
    subs = ()
    def reset(self):
        # extend (called from __init__ in ancestor)
        # Reset all data attributes
        self.verbatim = 0
        BaseHTMLProcessor.reset(self)
    def start_pre(self, attrs):
        # called for every <pre> tag in HTML source
        # Increment verbatim mode count, then handle tag like normal
        self.verbatim += 1
        self.unknown_starttag("pre", attrs)
    def end_pre(self):
        # called for every </pre> tag in HTML source
        # Decrement verbatim mode count
        self.unknown_endtag("pre")
        self.verbatim -= 1
    def handle_data(self, text):
        # override
        # called for every block of text in HTML source
        # If in verbatim mode, save text unaltered;
        # otherwise process the text with a series of substitutions
        self.pieces.append(self.verbatim and text or self.process(text))
    def process(self, text):
        # called from handle_data
        # Process text block by performing series of regular expression
        # substitutions (actual substitions are defined in descendant)
        for fromPattern, toPattern in self.subs:
            text = re.sub(fromPattern, toPattern, text)
        return text
class ChefDialectizer(Dialectizer):
    """convert HTML to Swedish Chef-speak
    based on the classic chef.x, copyright (c) 1992, 1993 John Hagerman
    """
    subs = ((r'a([nu])', r'u\1'),
            (r'A([nu])', r'U\1'),
            (r'a\B', r'e'),
            (r'A\B', r'E'),
            (r'en\b', r'ee'),
            (r'\Bew', r'oo'),
            (r'\Be\b', r'e-a'),
            (r'\be', r'i'),
            (r'\bE', r'I'),
            (r'\Bf', r'ff'),
            (r'\Bir', r'ur'),
            (r'(\w*?)i(\w*?)$', r'\1ee\2'),
            (r'\bow', r'oo'),
            (r'\bo', r'oo'),
            (r'\bO', r'Oo'),
            (r'the', r'zee'),
            (r'The', r'Zee'),
            (r'th\b', r't'),
            (r'\Btion', r'shun'),
            (r'\Bu', r'oo'),
            (r'\BU', r'Oo'),
            (r'v', r'f'),
            (r'V', r'F'),
            (r'w', r'w'),
            (r'W', r'W'),
            (r'([a-z])[.]', r'\1.  Bork Bork Bork!'))
class FuddDialectizer(Dialectizer):
    """convert HTML to Elmer Fudd-speak"""
    subs = ((r'[rl]', r'w'),
            (r'qu', r'qw'),
            (r'th\b', r'f'),
            (r'th', r'd'),
            (r'n[.]', r'n, uh-hah-hah-hah.'))
class OldeDialectizer(Dialectizer):
    """convert HTML to mock Middle English"""
    subs = ((r'i([bcdfghjklmnpqrstvwxyz])e\b', r'y\1'),
            (r'i([bcdfghjklmnpqrstvwxyz])e', r'y\1\1e'),
```

```
            (r'ick\b', r'yk'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'e[ea]([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'([bcdfghjklmnpqrstvwxyz])y', r'\1ee'),
            (r'([bcdfghjklmnpqrstvwxyz])er', r'\1re'),
            (r'([aeiou])re\b', r'\1r'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'i\1e'),
            (r'tion\b', r'cioun'),
            (r'ion\b', r'ioun'),
            (r'aid', r'ayde'),
            (r'ai', r'ey'),
            (r'ay\b', r'y'),
            (r'ay', r'ey'),
            (r'ant', r'aunt'),
            (r'ea', r'ee'),
            (r'oa', r'oo'),
            (r'ue', r'e'),
            (r'oe', r'o'),
            (r'ou', r'ow'),
            (r'ow', r'ou'),
            (r'\bhe', r'hi'),
            (r've\b', r'veth'),
            (r'se\b', r'e'),
            (r"'s\b", r'es'),
            (r'ic\b', r'ick'),
            (r'ics\b', r'icc'),
            (r'ical\b', r'ick'),
            (r'tle\b', r'til'),
            (r'll\b', r'l'),
            (r'ould\b', r'olde'),
            (r'own\b', r'oune'),
            (r'un\b', r'onne'),
            (r'rry\b', r'rye'),
            (r'est\b', r'este'),
            (r'pt\b', r'pte'),
            (r'th\b', r'the'),
            (r'ch\b', r'che'),
            (r'ss\b', r'sse'),
            (r'([wybdp])\b', r'\1e'),
            (r'([rnt])\b', r'\1\1e'),
            (r'from', r'fro'),
            (r'when', r'whan'))
def translate(url, dialectName="chef"):
    """fetch URL and translate using dialect
    dialect in ("chef", "fudd", "olde")"""
    import urllib
    sock = urllib.urlopen(url)
    htmlSource = sock.read()
    sock.close()
    parserName = "%sDialectizer" % dialectName.capitalize()
    parserClass = globals()[parserName]
    parser = parserClass()
    parser.feed(htmlSource)
    parser.close()
    return parser.output()
def test(url):
    """test all dialects against URL"""
    for dialect in ("chef", "fudd", "olde"):
        outfile = "%s.html" % dialect
        fsock = open(outfile, "wb")
        fsock.write(translate(url, dialect))
        fsock.close()
        import webbrowser
        webbrowser.open_new(outfile)
if __name__ == "__main__":
    test("http://diveintopython.org/odbchelper_list.html")
```

例 8.3. `dialect.py` 的输出结果

运行这个脚本会将 第 3.2 节 “List 介绍” 转换成模仿瑞典厨师用语 (mock Swedish Chef-speak) (来自 The Muppets)、模仿埃尔默唠叨者用语 (mock Elmer Fudd-speak) (来自 Bugs Bunny 卡通画) 和模仿中世纪英语 (mock Middle English) (零散地来源于乔叟的《坎特伯雷故事集》)。如果您查看输出页面的 HTML 源代码，您会发现所有的 HTML 标记和属性没有改动，但是在标记之间的文本被转换成模仿语言了。如果您观查得更仔细些，您会发现，实际上，仅有标题和段落被转换了；代码列表和屏幕例子没有改动。

```
<div class="abstract">
<p>Lists awe <span class="application">Pydon</span>'s wowkhowse datatype.
If youw onwy expewience wif wists is awways in
<span class="application">Visuaw Basic</span> ow (God fowbid) de datastowe
in <span class="application">Powewbuiwdew</span>, bwace youwsewf fow
<span class="application">Pydon</span> wists.</p>
</div>
```

8.2. `sgmllib.py` 介绍

HTML 处理分成三步：将 HTML 分解成它的组成片段，对片段进行加工，接着将片段再重新合成 HTML。第一步是通过 `sgmllib.py` 来完成的，它是标准 Python 库的一部分。

理解本章的关键是要知道 HTML 不只是文本，更是结构化文本。这种结构来源于开始与结束标记的或多或少分级序列。通常您并不以这种方式处理 HTML ，而是以文本方式 在一个文本编辑中对其进行处理，或以可视的方式 在一个浏览器中进行浏览或页面编辑工具中进行编辑。 `sgmllib.py` 表现出了 HTML 的结构。

`sgmllib.py` 包含一个重要的类： `SGMLParser` 。 `SGMLParser` 将 HTML 分解成有用的片段，比如开始标记和结束标记。在它成功地分解出某个数据为一个有用的片段后，它会根据所发现的数据，调用一个自身内部的方法。为了使用这个分析器，您需要子类化 `SGMLParser` 类，并且覆盖这些方法。这就是当我说它表示了 HTML 结构 的意思：HTML 的结构决定了方法调用的次序和传给每个方法的参数。

`SGMLParser` 将 HTML 分析成 8 类数据，然后对每一类调用单独的方法：

开始标记 (Start tag)

是开始一个块的 HTML 标记，像 `<html>` 、 `<head>` 、 `<body>` 或 `<pre>` 等，或是一个独一的标记，像 `<br>` 或 `<img>` 等。当它找到一个开始标记 *`tagname`* ， `SGMLParser` 将查找名为 `start_` tagname `_` 或 `do_` tagname `_` 的方法。例如，当它找到一个 `<pre>` 标记，它将查找一个 `start_pre` 或 `do_pre` 的方法。如果找到了， `SGMLParser` 会使用这个标记的属性列表来调用这个方法；否则，它用这个标记的名字和属性列表来调用 `unknown_starttag` 方法。

结束标记 (End tag)

是结束一个块的 HTML 标记，像 `</html>` 、 `</head>` 、 `</body>` 或 `</pre>` 等。当找到一个结束标记时， `SGMLParser` 将查找名为 `end_` tagname `_` 的方法。如果找到， `SGMLParser` 调用这个方法，否则它使用标记的名字来调用 `unknown_endtag` 。

字符引用 (Character reference)

用字符的十进制或等同的十六进制来表示的转义字符，像 ` ` 。当找到， `SGMLParser` 使用十进制或等同的十六进制字符文本来调用 `handle_charref` 。

实体引用 (Entity reference)

HTML 实体，像 `©` 。当找到， `SGMLParser` 使用 HTML 实体的名字来调用 `handle_entityref` 。

注释 (Comment)

HTML 注释，包括在 `<!-- ... -->` 之间。当找到，`SGMLParser` 用注释内容来调用 `handle_comment` 。

处理指令 (Processing instruction)

HTML 处理指令，包括在 `<? ... >` 之间。当找到，`SGMLParser` 用处理指令内容来调用 `handle_pi` 。

声明 (Declaration)

HTML 声明，如 `DOCTYPE` ，包括在 `<! ... >` 之间。当找到，`SGMLParser` 用声明内容来调用 `handle_decl` 。

文本数据 (Text data)

文本块。不满足其它 7 种类别的任何东西。当找到，`SGMLParser` 用文本来调用 `handle_data` 。

> 重要
> Python 2.0 存在一个 bug，即 `SGMLParser` 完全不能识别声明 (`handle_decl` 永远不会调用)，这就意味着 `DOCTYPE` 被静静地忽略掉了。这个错误在 Python 2.1 中改正了。

`sgmllib.py` 所附带的一个测试套件举例说明了这一点。您可以运行 `sgmllib.py` ，在命令行下传入一个 HTML 文件的名字，然后它会在分析标记和其它元素的同时将它们打印出来。它的实现是通过子类化 `SGMLParser` 类，然后定义 `unknown_starttag` ，`unknown_endtag` ，`handle_data` 和其它方法来实现的。这些方法简单地打印出它们的参数。

> 提示
> 在 Windows 下的 ActivePython IDE 中，您可以在 “Run script” 对话框中指定命令行参数。用空格将多个参数分开。

例 8.4. `sgmllib.py` 的样例测试

下面是一个片段，来自本书的 HTML 版本的目录，toc.html。当然，您的存储路径可能与我的有所不同。 (如果您还没有下载本书的 HTML 版本，可以从 http://diveintopython.org/ 下载。

```
c:\python23\lib> type "c:\downloads\diveintopython\html\toc\index.html"
 <!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
      <title>Dive Into Python</title>
      <link rel="stylesheet" href="diveintopython.css" type="text/css">
... 略 ...
```

通过 `sgmllib.py` 的测试套件来运行它，会得到如下的输出结果:

```
c:\python23\lib> python sgmllib.py "c:\downloads\diveintopython\html\toc\index.html"
data: '\n\n'
start tag: <html lang="en" >
data: '\n   '
start tag: <head>
data: '\n      '
start tag: <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" >
data: '\n   \n      '
start tag: <title>
data: 'Dive Into Python'
end tag: </title>
data: '\n      '
start tag: <link rel="stylesheet" href="diveintopython.css" type="text/css" >
data: '\n      '
... 略 ...
```

下面是本章其它部分的路标：

- 子类化 `SGMLParser` 来创建从 HTML 文档中抽取感兴趣的数据的类。
- 子类化 `SGMLParser` 来创建 `BaseHTMLProcessor`，它覆盖了所有8个处理方法，然后使用它们从片段中重建原始的 HTML。
- 子类化 `BaseHTMLProcessor` 来创建 `Dialectizer`，它增加了一些方法，专门用来处理指定的 HTML 标记，然后覆盖了 `handle_data` 方法，提供了用来处理 HTML 标记之间文本块的框架。
- 子类化 `Dialectizer` 来创建定义了文本处理规则的类。这些规则被 `Dialectizer.handle_data` 使用。
- 编写一个测试套件，它可以从 `http://diveintopython.org/` 处抓取一个真正的 web 页面，然后处理它。

继续阅读本章，您还可以学习到有关 `locals`、`globals` 和基于 dictionary 的字符串格式化的内容。

8.3. 从 HTML 文档中提取数据

为了从 HTML 文档中提取数据，将 `SGMLParser` 类进行子类化，然后对想要捕捉的标记或实体定义方法。

从 HTML 文档中提取数据的第一步是得到某个 HTML 文件。如果在您的硬盘里存放着 HTML 文件，您可以使用处理文件的函数将它读出来，但是真正有意思的是从实际的网页得到 HTML。

例 8.5. `urllib` 介绍

```
>>> import urllib
>>> sock = urllib.urlopen("http://diveintopython.org/")
>>> htmlSource = sock.read()
>>> sock.close()
>>> print htmlSource
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html
      <meta http-equiv='Content-Type' content='text/html; charset=ISO-8859-1'>
   <title>Dive Into Python</title>
<link rel='stylesheet' href='diveintopython.css' type='text/css'>
<link rev='made' href='mailto:mark@diveintopython.org'>
<meta name='keywords' content='Python, Dive Into Python, tutorial, object-oriented, progr
<meta name='description' content='a free Python tutorial for experienced programmers'>
</head>
<body bgcolor='white' text='black' link='#0000FF' vlink='#840084' alink='#0000FF'>
<table cellpadding='0' cellspacing='0' border='0' width='100%'>
<tr><td class='header' width='1%' valign='top'>diveintopython.org</td>
<td width='99%' align='right'><hr size='1' noshade></td></tr>
<tr><td class='tagline' colspan='2'>Python for experienced programmers</td
[...略...]
```

[1]	`urllib` 模块是标准 Python 库的一部分。它包含了一些函数，可以从基于互联网的 URL (主要指网页) 来获取信息并且真正取回数据。
[2]	`urllib` 模块最简单的使用是提取用 `urlopen` 函数取回的网页的整个文本。打开一个 URL 同打开一个文件相似。 `urlopen` 的返回值是像文件一样的对象，它具有一个文件对象一样的方法。
[3]	使用由 `urlopen` 所返回的类文件对象所能做的最简单的事情就是 `read` ，它可以将网页的整个 HTML 读到一个字符串中。这个对象也支持 `readlines` 方法，这个方法可以将文本按行放入一个列表中。
[4]	当用完这个对象，要确保将它 `close` ，就如同一个普通的文件对象。
[5]	现在我们将 `http://diveintopython.org/` 主页的完整的 HTML 保存在一个字符串中了，接着我们将分析它。

例 8.6. `urllister.py` 介绍

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
 from sgmllib import SGMLParser
class URLLister(SGMLParser):
    def reset(self):
        SGMLParser.reset(self)
        self.urls = []
    def start_a(self, attrs):
        href = [v for k, v in attrs if k=='href']
        if href:
            self.urls.extend(href)
```

[1]	`reset` 由 `SGMLParser` 的 `__init__` 方法来调用，也可以在创建一个分析器实例时手工来调用。所以如果您需要做初始化，在 `reset` 中去做，而不要在 `__init__` 中做。这样当某人重用一个分析器实例时，可以正确地重新初始化。
[2]	只要找到一个 `<a>` 标记， `start_a` 就会由 `SGMLParser` 进行调用。这个标记可以包含一个 `href` 属性，或者包含其它的属性，如 `name` 或 `title` 。 `attrs` 参数是一个 tuple 的 list， `[(_attribute_, _value_), (_attribute_, _value_), ...]` 。或者它可以只是一个有效的 HTML 标记 `<a>` (尽管无用)，这时 `attrs` 将是个空 list。
[3]	我们可以通过一个简单的多变量 list 映射来查找这个 `<a>` 标记是否拥有一个 `href` 属性。
[4]	像 `k=='href'` 的字符串比较是区分大小写的，但是这里是安全的。因为 `SGMLParser` 会在创建 `attrs` 时将属性名转化为小写。

例 8.7. 使用 `urllister.py`

```
>>> import urllib, urllister
>>> usock = urllib.urlopen("http://diveintopython.org/")
>>> parser = urllister.URLLister()
>>> parser.feed(usock.read())
>>> usock.close()
>>> parser.close()
>>> for url in parser.urls: print url
toc/index.html
#download
#languages
toc/index.html
appendix/history.html
download/diveintopython-html-5.0.zip
download/diveintopython-pdf-5.0.zip
download/diveintopython-word-5.0.zip
download/diveintopython-text-5.0.zip
download/diveintopython-html-flat-5.0.zip
download/diveintopython-xml-5.0.zip
download/diveintopython-common-5.0.zip
...略...
```

[1]	调用定义在 `SGMLParser` 中的 `feed` 方法，将 HTML 内容放入分析器中。 [4] 这个方法接收一个字符串，这个字符串就是 `usock.read()` 所返回的。
[2]	像处理文件一样，一旦处理完毕，您应该 `close` 您的 URL 对象。
[3]	您也应该 `close` 您的分析器对象，但出于不同的原因。 `feed` 方法不保证对传给它的全部 HTML 进行处理，它可能会对其进行缓冲处理，等待接收更多的内容。只要没有更多的内容，就应调用 `close` 来刷新缓冲区，并且强制所有内容被完全处理。
[4]	一旦分析器被 `close` ，分析过程也就结束了。 `parser.urls` 中包含了在 HTML 文档中所有的链接 URL。(如果当您读到此处发现输出结果不一样，那是因为下载了本书的更新版本。)

Footnotes

[4] 像 `SGMLParser` 这样的分析器，技术术语叫做消费者 *(consumer)*。它消费 HTML，并且拆分它。也许因此就选择了 `feed` 这个名字，以便同消费者 这个主题相适应。就个人来说，它让我想象在动物园看展览。里面有一个黑漆漆的兽穴，没有树，没有植物，没有任何生命的迹象。但只要您非常安静地站着，尽可能靠近着瞧，您会看到在远处的角落里有两只明眸在盯着您。但是您会安慰自己那不过是心理作用。唯一知道兽穴里并不是空无一物的方法，就是在栅栏上有一个不明显的标记，上面写着 “禁止给分析器喂食”。但也许只有我这么想，不管怎么样，这种心理想象很有意思。

8.4. `BaseHTMLProcessor.py` 介绍

`SGMLParser` 自身不会产生任何结果。它只是分析，分析，再分析，对于它找到的有趣的东西会调用相应的一个方法，但是这些方法什么都不做。`SGMLParser` 是一个 HTML 消费者 *(consumer)*：它接收 HTML，将其分解成小的、结构化的小块。正如您所看到的，在前一节中，您可以定义 `SGMLParser` 的子类，它可以捕捉特别标记和生成有用的东西，如一个网页中所有链接的一个列表。现在我们将沿着这条路更深一步。我们要定义一个可以捕捉 `SGMLParser` 所丢出来的所有东西的一个类，接着重建整个 HTML 文档。用技术术语来说，这个类将是一个 HTML 生产者 *(producer)*。

`BaseHTMLProcessor` 子类化 `SGMLParser`，并且提供了全部的 8 个处理方法：`unknown_starttag`、`unknown_endtag`、`handle_charref`、`handle_entityref`、`handle_com`、`handle_pi`、`handle_decl` 和 `handle_data`。

例 8.8. `BaseHTMLProcessor` 介绍

```
class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        self.pieces = []
        SGMLParser.reset(self)
    def unknown_starttag(self, tag, attrs):
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
    def unknown_endtag(self, tag):
        self.pieces.append("</%(tag)s>" % locals())
    def handle_charref(self, ref):
        self.pieces.append("&#%(ref)s;" % locals())
    def handle_entityref(self, ref):
        self.pieces.append("&%(ref)s" % locals())
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")
    def handle_data(self, text):
        self.pieces.append(text)
    def handle_comment(self, text):
        self.pieces.append("<!--%(text)s-->" % locals())
    def handle_pi(self, text):
        self.pieces.append("<?%(text)s>" % locals())
    def handle_decl(self, text):
        self.pieces.append("<!%(text)s>" % locals())
```

[1]	`reset` 由 `SGMLParser.__init__` 来调用。在调用父类方法之前将 `self.pieces` 初始化为空列表。 `self.pieces` 是一个数据属性，将用来保存将要构造的 HTML 文档的片段。每个处理器方法都将重构 `SGMLParser` 所分析出来的 HTML，并且每个方法将生成的字符串追加到 `self.pieces` 之后。注意， `self.pieces` 是一个 list。也许您想将它定义为一个字符串，然后不停地将每个片段追加到它的后面。这样做是可以的，但是 Python 在处理 list 方面效率更高一些。 [5]
[2]	因为 `BaseHTMLProcessor` 没有为特别标记定义方法 (如在 `URLLister` 中的 `start_a` 方法)， `SGMLParser` 将对每一个开始标记调用 `unknown_starttag` 方法。这个方法接收标记 (`tag`) 和属性的名字/值对的 list(`attrs`) 两参数，重新构造初始的 HTML，接着将结果追加到 `self.pieces` 后。 这里的字符串格式化有些陌生，我们将留到下一节再说明。
[3]	重构结束标记要简单得多，只是使用标记名字，把它包在 </...> 括号中。
[4]	当 `SGMLParser` 找到一个字符引用时，会用原始的引用来调用 `handle_charref` 。如果 HTML 文档包含 ` ` 这个引用， `ref` 将为 `160` 。重构原始的完整的字符引用只要将 `ref` 包装在 `&#...;` 字符中间。
[5]	实体引用同字符引用相似，但是没有#号。重建原始的实体引用只要将 `ref` 包装在 `&...;` 字符串中间。(实际上，一位博学的读者曾经向我指出，除些之外还稍微有些复杂。仅有某种标准的 HTML 实体以一个分号结束；其它看上去差不多的实体并不如此。幸运的是，标准 HTML 实体集已经定义在 Python 的一个叫做 `htmlentitydefs` 的模块中了。从而引出额外的 `if` 语句。)
[6]	文本块则简单地不经修改地追加到 `self.pieces` 后。
[7]	HTML 注释包装在 <!--...--> 字符中。
[8]	处理指令包装在 <?...> 字符中。

> 重要
>
> HTML 规范要求所有非 HTML (像客户端的 JavaScript) 必须包括在 HTML 注释中，但不是所有的页面都是这么做的 (而且所有的最新的浏览器也都容许不这样做) 。 `BaseHTMLProcessor` 不允许这样，如果脚本嵌入得不正确，它将被当作 HTML 一样进行分析。例如，如果脚本包含了小于和等于号， `SGMLParser` 可能会错误地认为找到了标记和属性。 `SGMLParser` 总是把标记名和属性名转换成小写，这样可能破坏了脚本，并且 `BaseHTMLProcessor` 总是用双引号来将属性封闭起来 (尽管原始的 HTML 文档可能使用单引号或没有引号) ，这样必然会破坏脚本。应该总是将您的客户端脚本放在 HTML 注释中进行保护。

例 8.9. `BaseHTMLProcessor` 输出结果

```
    def output(self):
        """Return processed HTML as a single string"""
        return "".join(self.pieces)
```

[1]	这是在 `BaseHTMLProcessor` 中的一个方法，它永远不会被父类 `SGMLParser` 所调用。因为其它的处理器方法将它们重构的 HTML 保存在 `self.pieces` 中，这个函数需要将所有这些片段连接成一个字符串。正如前面提到的，Python 在处理列表方面非常出色，但对于字符串处理就逊色了。所以我们只有在某人确实需要它时才创建完整的字符串。
[2]	如果您愿意，也可以换成使用 `string` 模块的 `join` 方法：`string.join(self.pieces, "")` 。

进一步阅读

- W3C 讨论了字符和实体引用。
- *Python Library Reference* 解答了您的怀疑，即 `htmlentitydefs` 模块的确名符其实。

Footnotes

[5] Python 处理 list 比字符串快的原因是：list 是可变的，但字符串是不可变的。这就是说向 list 进行追加只是增加元素和修改索引。因为字符串在创建之后不能被修改，像 `s = s + newpiece` 这样的代码将会从原值和新片段的连接结果中创建一个全新的字符串，然后丢弃原来的字符串。这样就需要大量昂贵的内存管理，并且随着字符串变长，所需要的开销也在增长。所以在一个循环中执行 `s = s + newpiece` 非常不好。用技术术语来说，向一个 list 追加 `n` 个项的代价为 `O(n)` ，而向一个字符串追加 `n` 个项的代价是 `O(n<sup>2</sup>)` 。

8.5. `locals` 和 `globals`

我们先偏离一下 HTML 处理的主题，讨论一下 Python 如何处理变量。Python 有两个内置的函数， `locals` 和 `globals` ，它们提供了基于 dictionary 的访问局部和全局变量的方式。

还记得 `locals` 吗？您第一次是在这里看到的：

```
def unknown_starttag(self, tag, attrs):
    strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
    self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
```

不，等等，此时您还不能理解 `locals` 。首先，您需要学习关于命名空间的知识。这很枯燥，但是很重要，因此要要耐心些。

Python 使用叫做名字空间的东西来记录变量的轨迹。名字空间只是一个 dictionary ，它的键字就是变量名，它的值就是那些变量的值。实际上，名字空间可以像 Python 的 dictionary 一样进行访问，一会儿我们就会看到。

在一个 Python 程序中的任何一个地方，都存在几个可用的名字空间。每个函数都有着自已的名字空间，叫做局部名字空间，它记录了函数的变量，包括函数的参数和局部定义的变量。每个模块拥有它自已的名字空间，叫做全局名字空间，它记录了模块的变量，包括函数、类、其它导入的模块、模块级的变量和常量。还有就是内置名字空间，任何模块均可访问它，它存放着内置的函数和异常。

当一行代码要使用变量 `x` 的值时，Python 会到所有可用的名字空间去查找变量，按照如下顺序：

1. 局部名字空间——特指当前函数或类的方法。如果函数定义了一个局部变量 `x`，或一个参数 `x`，Python 将使用它，然后停止搜索。
2. 全局名字空间——特指当前的模块。如果模块定义了一个名为 `x` 的变量，函数或类，Python 将使用它然后停止搜索。
3. 内置名字空间——对每个模块都是全局的。作为最后的尝试，Python 将假设 `x` 是内置函数或变量。

如果 Python 在这些名字空间找不到 `x`，它将放弃查找并引发一个 `NameError` 异常，同时传递 `There is no variable named 'x'` 这样一条信息，回到 例 3.18 “引用未赋值的变量”，您会看到一路上都有这样的信息。但是您并没有体会到 Python 在给出这样的错误之前做了多少的努力。

> 重要
> Python 2.2 引入了一种略有不同但重要的改变，它会影响名字空间的搜索顺序：嵌套的作用域。 在 Python 2.2 版本之前，当您在一个嵌套函数或 `lambda` 函数中引用一个变量时，Python 会在当前 (嵌套的或 `lambda`) 函数的名字空间中搜索，然后在模块的名字空间。Python 2.2 将只在当前 (嵌套的或 `lambda`) 函数的名字空间中搜索，然后是在父函数的名字空间 中搜索，接着是模块的名字空间中搜索。Python 2.1 可 以两种方式工作，缺省地，按 Python 2.0 的方式工作。但是您可以把下面一行代码增加到您的模块头部，使您的模块工作起来像 Python 2.2 的方式：
>
> ```
> from __future__ import nested_scopes
> ```

您是否为此而感到困惑？不要灰心！我敢说这一点非常酷。像 Python 中的许多事情一样，名字空间在运行时直接可以访问。怎么样？不错吧，局部名字空间可以通过内置的 `locals` 函数来访问。全局 (模块级别) 名字空间可以通过内置的 `globals` 函数来访问。

例 8.10. `locals` 介绍

```
>>> def foo(arg):
... x = 1
... print locals()
...
>>> foo(7)
{'arg': 7, 'x': 1}
>>> foo('bar')
{'arg': 'bar', 'x': 1}
```

[1]	函数 `foo` 在它的局部名字空间中有两个变量：`arg` (它的值是被传入函数的) 和 `x` (它是在函数里定义的)。
[2]	`locals` 返回一个名字/值对的 dictionary。这个 dictionary 的键字是字符串形式的变量名字，dictionary 的值是变量的实际值。所以用 `7` 来调用 `foo`，会打印出包含函数两个局部变量的 dictionary：`arg` (`7`) 和 `x` (`1`)。
[3]	回想一下，Python 有动态数据类型，所以您可以非常容易地传递给 `arg` 一个字符串，这个函数 (和对 `locals` 的调用) 将仍然很好的工作。`locals` 可以用于所有类型的变量。

`locals` 对局部 (函数) 名字空间做了些什么，`globals` 就对全局 (模块) 名字空间做了什么。然而 `globals` 更令人兴奋，因为一个模块的名字空间是更令人兴奋的。[6] 模块的名字空间不仅仅包含了模块级的变量和常量，还包括了所有在模块中定义的函数和类。除此以外，它还包括了任何被导入到模块中的东西。

回想一下 `from _module_ import` 和 `import _module_` 之间的不同。使用 `import _module_`，模块自身被导入，但是它保持着自已的名字空间，这就是为什么您需要使用模块名来访问它的函数或属性：`_module_._function_` 的原因。但是使用 `from _module_ import`，实际上是从

另一个模块中将指定的函数和属性导入到您自己的名字空间，这就是为什么您可以直接访问它们却不需要引用它们所来源的模块。使用 `globals` 函数，您会真切地看到这一切的发生。

例 8.11. `globals` 介绍

看看下面列出的在文件 `BaseHTMLProcessor.py` 尾部的代码块：

```
if __name__ == "__main__":
    for k, v in globals().items():
        print k, "=", v
```

[1]	不要被吓坏了，想想以前您已经全部都看到过了。`globals` 函数返回一个 dictionary，我们使用 `items` 方法和多变量赋值来遍历 dictionary。在这里唯一的新东西就是 `globals` 函数。

现在从命令行运行这个脚本，会得到下面的输出 (注意您的输出可能有略微的不同，这依赖于您的系统平台和所安装的 Python 版本)：

```
c:\docbook\dip\py> python BaseHTMLProcessor.py
```

```
SGMLParser = sgmllib.SGMLParser
htmlentitydefs = <module 'htmlentitydefs' from 'C:\Python23\lib\htmlentitydefs.py'>
BaseHTMLProcessor = __main__.BaseHTMLProcessor
__name__ = __main__
... rest of output omitted for brevity...
```

[1]	我们使用了 `from _module_ import` 把 `SGMLParser` 从 `sgmllib` 中导入。也就是说它被直接导入到我们的模块名字空间了，就是这样。
[2]	把上面的例子和 `htmlentitydefs` 对比一下，它是用 `import` 被导入的。也就是说 `htmlentitydefs` 模块本身被导入了名字空间，但是定义在 `htmlentitydefs` 之中的 `entitydefs` 变量却没有。
[3]	这个模块只定义一个类，`BaseHTMLProcessor`，不错。注意这儿的值就是类本身，不是一个特别的类实例。
[4]	记得 `if __name__` 技巧吗？当运行一个模块时 (相对于从另外一个模块中导入而言)，内置的 `__name__` 是一个特殊值 `__main__`。因为我们是把这个模块当作脚本从命令来运行的，故 `__name__` 值为 `__main__`，这就是为什么我们这段简单地打印 `globals` 的代码可以执行的原因。

> 注意
>
> 使用 `locals` 和 `globals` 函数，通过提供变量的字符串名字您可以动态地得到任何变量的值。这种方法提供了这样的功能：`getattr` 函数允许您通过提供函数的字符串名来动态地访问任意的函数。

在 `locals` 与 `globals` 之间有另外一个重要的区别，您应该在它困扰您之前就了解它。它无论如何都会困扰您的，但至少您还会记得曾经学习过它。

例 8.12. `locals` 是只读的， `globals` 不是

```
 def foo(arg):
    x = 1
    print locals()
    locals()["x"] = 2
    print "x=",x
z = 7
print "z=",z
foo(3)
globals()["z"] = 8      print "z=",z
```

[1]	因为使用 `3` 来调用 `foo` ，会打印出 `{'arg': 3, 'x': 1}` 。这个应该没什么奇怪的。
[2]	`locals` 是一个返回 dictionary 的函数，这里您在 dictionary 中设置了一个值。您可能认为这样会改变局部变量 `x` 的值为 `2` ，但并不会。 `locals` 实际上没有返回局部名字空间，它返回的是一个拷贝。所以对它进行改变对局部名字空间中的变量值并无影响。
[3]	这样会打印出 `x= 1` ，而不是 `x= 2` 。
[4]	在有了对 `locals` 的经验之后，您可能认为这样不会 改变 `z` 的值，但是可以。由于 Python 在实现过程中内部有所区别 (关于这些区别我宁可不去研究，因为我自已还没有完全理解) ， `globals` 返回实际的全局名字空间，而不是一个拷贝：与 `locals` 的行为完全相反。所以对 `globals` 所返回的 dictionary 的任何的改动都会直接影响到全局变量。
[5]	这样会打印出 `z= 8` ，而不是 `z= 7` 。

Footnotes

[6] 我没有说得太多吧。

8.6. 基于 dictionary 的字符串格式化

为什么学习 `locals` 和 `globals`？因为接下来就可以学习关于基于 dictionary 的字符串格式化。或许您还能记起，字符串格式化提供了一种将值插入字符串中的一种便捷的方法。值被列在一个 tuple 中，按照顺序插入到字符串中每个格式化标记所在的位置上。尽管这种做法效率高，但还不是最容易阅读的代码，特别是当插入多个值的时候。仅用眼看一遍字符串，您不能马上就明白结果是什么；您需要经常地在字符串和值的 tuple 之间进行反复查看。

有另外一种字符串格式化的形式，它使用 dictionary 而不是值的 tuple。

例 8.13. 基于 dictionary 的字符串格式化介绍

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> "%(pwd)s" % params
'secret'
>>> "%(pwd)s is not a good password for %(uid)s" % params
'secret is not a good password for sa'
>>> "%(database)s of mind, %(database)s of body" % params
'master of mind, master of body'
```

[1]	这种字符串格式化形式不用显式的值的 tuple，而是使用一个 dictionary，`params`。并且标记也不是在字符串中的一个简单 `%s`，而是包含了一个用括号包围起来的名字。这个名字是 `params` dictionary 中的一个键字，所以 `%(pwd)s` 标记被替换成相应的值 `secret`。
[2]	基于 dictionary 的字符串格式化可用于任意数量的有名的键字。每个键字必须在一个给定的 dictionary 中存在，否则这个格式化操作将失败并引发一个 `KeyError` 的异常。
[3]	您甚至可以两次指定同一键字，每个键字出现之处将被同一个值所替换。

那么为什么您偏要使用基于 dictionary 的字符串格式化呢？的确，仅为了进行字符串格式化，就事先创建一个有键字和值的 dictionary 看上去的确有些小题大作。它的真正最大用处是当您碰巧已经有了像 `locals` 一样的有意义的键字和值的 dictionary 的时候。

例 8.14. `BaseHTMLProcessor.py` 中的基于 dictionary 的字符串格式化

```
    def handle_comment(self, text):
        self.pieces.append("<!--%(text)s-->" % locals())
```

[1]	使用内置的 `locals` 函数是最普通的基于 dictionary 的字符串格式化的应用。这就是说您可以在您的字符串 (本例中是 `text` ，它作为一个参数传递给类方法) 中使用局部变量的名字，并且每个命名的变量将会被它的值替换。如果 `text` 是 `'Begin page footer'` ，字符串格式化 `"<!--%(text)s-->" % locals()` 将得到字符串 `'<!--Begin page footer-->'` 。

例 8.15. 基于 dictionary 的字符串格式化的更多内容

```
def unknown_starttag(self, tag, attrs):
    strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
    self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
```

[1]	当这个模块被调用时， `attrs` 是一个键/值 tuple 的 list，就像一个 dictionary 的 `items` 。这就意味着我们可以使用多变量赋值来遍历它。到现在这将是一种熟悉的模式，但是这里有很多东西，让我们分开来看：1. 假设 `attrs` 是 `[('href', 'index.html'), ('title', 'Go to home page')]` 。2. 在这个列表解析的第一轮循环中， `key` 将为 `'href'` ， `value` 将为 `'index.html'` 。3. 字符串格式化 `' %s="%s"' % (key, value)` 将生成 `' href="index.html"'` 。这个字符串就作为这个列表解析返回值的第一个元素。4. 在第二轮中， `key` 将为 `'title'` ， `value` 将为 `'Go to home page'` 。5. 字符串格式化将生成 `' title="Go to home page"'` 。6. 这个 list 理解返回两个生成的字符串 list，并且 `strattrs` 将把这个 list 的两个元素连接在一起形成 `' href="index.html" title="Go to home page"'` 。
[2]	现在，使用基于 dictionary 的字符串格式化，我们将 `tag` 和 `strattrs` 的值插入到一个字符串中。所以，如果 `tag` 是 `'a'` ，最终的结果会是 `'<a href="index.html" title="Go to home page">'` ，并且这就是追加到 `self.pieces` 后面的东西。

> 重要
>
> 使用 `locals` 来应用基于 dictionary 的字符串格式化是一种方便的作法，它可以使复杂的字符串格式化表达式更易读。但它需要花费一定的代价。在调用 `locals` 方面有一点性能上的问题，这是由于 `locals` 创建了局部名字空间的一个拷贝引起的。

8.7. 给属性值加引号

在 comp.lang.python 上的一个常见问题是 “我有一些 HTML 文档，属性值没有用引号括起来，并且我想将它们全部括起来，我怎么才能实现它呢？” [7] (一般这种事情的出现是由于一个项目经理加入到一个大的项目中来，而他又抱着 HTML 是一种标记语言的教条，要求所有的页面必须能够通过 HTML 校验器的验证。而属性值没有被引号括起来是一种常见的对 HTML 规范的违反。) 不管什么原因，未括起来的属性值通过将 HTML 送进 `BaseHTMLProcessor` 可以容易地修复。

`BaseHTMLProcessor` 消费 (consume) HTML (因为它是从 `SGMLParser` 派生来的) 并生成等价的 HTML。但是这个 HTML 输出与输入的并不一样。标记和属性名最终会转化为小写字母，即使它们可能以大写字母开始或是大小写的混合形式。属性值将被双引号引起来，即使它们原来可能是用单引号括起来的或根本没有括起来。这就是最后我们可以受益的边际效应。

例 8.16. 给属性值加引号

```
>>> htmlSource = """
... <html>
... <head>
... <title>Test page</title>
... </head>
... <body>
... <ul>
... <li><a href=index.html>Home</a></li>
... <li><a href=toc.html>Table of contents</a></li>
... <li><a href=history.html>Revision history</a></li>
... </body>
... </html>
... """
>>> from BaseHTMLProcessor import BaseHTMLProcessor
>>> parser = BaseHTMLProcessor()
>>> parser.feed(htmlSource)
>>> print parser.output()
<html>
<head>
<title>Test page</title>
</head>
<body>
<ul>
<li><a href="index.html">Home</a></li>
<li><a href="toc.html">Table of contents</a></li>
<li><a href="history.html">Revision history</a></li>
</body>
</html>
```

[1]	请注意，在 `<a>` 标记中的 `href` 属性值没有被适当地括起来 (还要注意，除了文档字符串之外，我们还将三重引号用到了 `doc string` 之外的其它地方，并且是不会少于直接在 IDE 中的使用。它们非常有用。)
[2]	装填分析器。
[3]	使用定义在 `BaseHTMLProcessor` 中的 `output` 函数，我们得到单个字符串的输出，并且属性值被完全括起来了。让我们想一下这里实际上发生了多少事：`SGMLParser` 分析整个 HTML 文档，将其分解为一片片的标记、引用、数据等等。`BaseHTMLProcessor` 使用这些元素来重新构造 HTML 的片段 (如果您想查看的话它们仍然保存在 `parser.pieces` 中) 。最后，我们调用 `parser.output` ，它将所有的 HTML 片段连接成一个字符串。

Footnotes

[7] 好吧，其实并不是那么普通的一个问题。在那不都是问 “我应该用何种编辑器来写 Python 代码？” (回答：Emacs) 或 “Python 比 Perl 是好还是坏？” (回答：“Perl 比 Python 差，因为人们想让它差的。” ——Larry Wall，1998 年 10 月 14 日) 但是关于 HTML 处理的问题，或者这种提法或者另一种提法，大约一个月就要出现一次，在这些问题之中，这个问题是最常见的一个。

8.8. dialect.py 介绍

Dialectizer 是 BaseHTMLProcessor 的简单 (和拙劣) 的派生类。它通过一系列的替换对文本块进行了处理，但是它确保在 <pre>`...`</pre> 块之间的任何东西不被修改地通过。

为了处理 <pre> 块，我们在 Dialectizer 中定义了两个方法：start_pre 和 end_pre 。

例 8.17. 处理特别标记

```
    def start_pre(self, attrs):
        self.verbatim += 1
        self.unknown_starttag("pre", attrs)
    def end_pre(self):
        self.unknown_endtag("pre")
        self.verbatim -= 1
```

[1]	每当 SGMLParser 在 HTML 源代码中发现一个 <pre> 时，都会调用 start_pre 。(马上我们就会确切地看到它是如何发生的。) 这个方法使用单个参数：attrs ，这个参数会包含标记的属性 (如果存在的话) 。 attrs 是一个键/值 tuple 的 list，就像 unknown_starttag 中所使用的。
[2]	在 reset 方法中，我们初始化了一个数据属性，它作为 <pre> 标记的一个计数器。每当我们找到一个 <pre> 标记，我们增加计数器的值；每当我们找到一个 </pre> 标记，我们将减少计数器的值。(我们本可以把它实现为一个标志，即或把它设为 1 ，或重置为 0 ，但这样做只是为了方便，并且这样做可以处理古怪 (但有可能) 的 <pre> 标记嵌套的情况。) 马上我们将会看到这个计数器是多么的好用。
[3]	不错，这就是我们对 <pre> 标记所做的唯一的特殊处理。现在我们将属性列表传给 unknown_starttag ，由它来进行缺省的处理。
[4]	每当 SGMLParser 找到一个 </pre> 标记时，会调用 end_pre 。因为结束标记不能包含属性，因此这个方法没有参数。
[5]	首先我们要进行缺省处理，就像其它结束标记做的一样。
[6]	其次我们将计数器减少，标记这个 <pre> 块已经被关闭了。

到了这个地方，有必要对 SGMLParser 更深入一层。我已经多次声明 (到目前为止您应已经把它做为信条了)，就是 SGMLParser 查找每一个标记并且如果存在特定的方法就调用它们。例如：我们刚刚看到处理 <pre> 和 </pre> 的 start_pre 和 end_pre 的定义。但这是如何发生的呢？嗯，也没什么神奇的，只不过是出色的 Python 编码。

例 8.18. `SGMLParser`

```
    def finish_starttag(self, tag, attrs):
        try:
            method = getattr(self, 'start_' + tag)
        except AttributeError:
            try:
                method = getattr(self, 'do_' + tag)
            except AttributeError:
                self.unknown_starttag(tag, attrs)
                return -1
            else:
                self.handle_starttag(tag, method, attrs)
                return 0
        else:
            self.stack.append(tag)
            self.handle_starttag(tag, method, attrs)
            return 1
    def handle_starttag(self, tag, method, attrs):
        method(attrs)
```

[1]	此处，`SGMLParser` 已经找到了一个开始标记，并且分析出属性列表。唯一要做的事情就是检查对于这个标记是否存在一个特别的处理方法，否则我们就应该求助于缺省方法 (`unknown_starttag`) 。
[2]	`SGMLParser` 的 “神奇” 之处除了我们的老朋友 `getattr` 之外就没有什么了。您以前可能没注意到，`getattr` 将查找定义在一个对象的继承者中或对象自身的方法。这里对象是 `self`，即当前实例。所以，如果 `tag` 是 `'pre'`，这里对 `getattr` 的调用将会在当前实例 (它是 `Dialectizer` 类的一个实例) 中查找一个名为 `start_pre` 的方法。
[3]	如果 `getattr` 所查找的方法在对象或它的任何继承者中不存在的话，它会引发一个 `AttributeError` 的异常。但没有关系，因为我们把对 `getattr` 的调用包装到一个 `try...except` 块中了，并且显式地捕捉 `AttributeError` 异常。
[4]	因为我们没有找到一个 `start_xxx` 方法，在放弃之前，我们将还要查找一个 `do_xxx` 方法。这个可替换的命名模式一般用于单独的标记，如 ` `，这些标记没有相应的结束标记。但是您可以使用任何一种模式，正如您看到的，`SGMLParser` 对每个标记尝试两次。(您不应该对相同的标记同时定义 `start_xxx` 和 `do_xxx` 处理方法，因为这样的话只有 `start_xxx` 方法会被调用。)
[5]	另一个 `AttributeError` 异常，它是说用 `do_xxx` 来调用 `getattr` 失败了。因为对同一个标记我们既没有找到 `start_xxx` 也没有找到 `do_xxx` 处理方法，这样我们捕捉到了异常并且求助于缺省方法：`unknown_starttag`。
[6]	记得吗？`try...except` 块可以有一个 `else` 子句，当在 `try...except` 块中没有异常被引发时，它将被调用。逻辑上，意味着我们确实找到了这个标记的 `do_xxx` 方法，所以我们将要调用它。
[7]	顺便说，不要为这些不同的返回值而担心；理论上他们有意义，但实际上它们没有任何用处。也不要担心 `self.stack.append(tag)`；`SGMLParser` 内部会知晓您的开始标记是否有合适的结束标记与之匹配，但是它不会对这些信息做任何操作。理论上，您能使用这个模块校验您的标记是否完全匹配，但是这或许没有多大价值，并且这样的内容已经超出了本章所要讨论的范畴。现在有您更需要担心的问题。
[8]	`start_xxx` 和 `do_xxx` 方法并不被直接调用；标记名、方法和属性被传给 `handle_starttag` 这个方法，以便继承者可以覆盖它，并改变全部 开始标记分发的方式。我们不需要控制这个层面，所以我们只让这个方法做它自已的事，就是用属性 list 来调用方法 (`start_xxx` 或 `do_xxx`) 。记住 `method` 是一个从 `getattr` 返回的函数，而函数是对象。(我知道您已经听腻了，我发誓，一旦我们停止寻找新的使用方法来为我们服务时，我就决不再提它了。) 这时，函数对象作为一个参数传入这个分发方法，这个方法反过来再调用这个函数。在这里，我们不需要知道函数是什么，叫什么名字，或是在哪时定义的；我们只需要知道用一个参数 `attrs` 调用它。

现在回到我们已经计划好的程序：`Dialectizer`。当我们跑题时，我们定义了特别的处理方法来处理 `<pre>` 和 `</pre>` 标记。还有一件事没有做，那就是用我们预定义的替换处理来处理文本块。为了实现它，我们需要覆盖 `handle_data` 方法。

例 8.19. 覆盖 `handle_data` 方法

```
def handle_data(self, text):
    self.pieces.append(self.verbatim and text or self.process(text))
```

[1]	`handle_data` 在调用时只使用一个参数：要处理的文本。
[2]	在祖先类 `BaseHTMLProcessor` 中， `handle_data` 方法只是将文本追加到输出缓冲区 `self.pieces` 之后。这里的逻辑稍微有点复杂。如果我们处于 <pre>`...`</pre> 块的中间， `self.verbatim` 将是大于 `0` 的某个值，接着我们想要将文本不作改动地传入输出缓冲区。否则，我们将调用另一个单独的方法来进行替换处理，然后将处理结果放入输出缓冲区中。在 Python 中，这是一个一行代码，它使用了 `and-or` 技巧。

我们已经接近了对 `Dialectizer` 的全面理解。唯一缺少的一个环节是文本替换的特性。如果您知道点 Perl，您就会知道当需要复杂的文本替换时，唯一有效的解决方法就是正则表达式。在 `dialect.py` 文件后面的几个类中定义了一连串的正则表达式来操作 HTML 标记中的文本。我们已经学习过了正则表达式中的所有字符。我们不必重复学习正则表达式的艰难历程了，不是吗？上帝知道我反正不需要。我想现在这章您已经学得差不多了。

8.9. 全部放在一起

到了将迄今为止我们已经学过并用得不错的东西放在一起的时候了。我希望您专心些。

例 8.20. `translate` 函数，第 1 部分

```
def translate(url, dialectName="chef"):
    import urllib
    sock = urllib.urlopen(url)
    htmlSource = sock.read()
    sock.close()
```

[1]	这个 `translate` 函数有一个可选参数 `dialectName`，它是一个字符串，指出我们将使用的方言。一会我们就会看到它是如何使用的。
[2]	嘿，等一下，在这个函数中有一个 `import` 语句！它在 Python 中完全合法。您已经习惯了在一个程序的前面看到 `import` 语句，它意味着导入的模块在程序的任何地方都是可用的。但您也可以在一个函数中导入模块，这意味着导入的模块只能在函数中使用。如果您有一个只能用在一个函数中的模块，这是一个简便的方法，使您的代码更模块化。(当发现您周末的加班已经变成了一个 800 行的艺术作品，并且决定将其分割成一打可重用的模块时，您会感谢它的。)
[3]	现在我们得到了给定的 URL 源文件。

例 8.21. `translate` 函数，第 2 部分：奇妙而又奇妙

```
    parserName = "%sDialectizer" % dialectName.capitalize()
    parserClass = globals()[parserName]
    parser = parserClass()
```

[1]	`capitalize` 是一个我们以前未曾见过的字符串方法；它只是将一个字符串的第一个字母变成大写，将其它的字母强制变成小写。再使用字符串格式化，我们就得到了一种方言的名字，并将它转化为了相应的方言变换器类的名字。如果 `dialectName` 是字符串 `'chef'`，`parserName` 将是字符串 `'ChefDialectizer'`。
[2]	我们有了一个字符串形式 (`parserName`) 的类名称，还有一个 dictionary (`globals` ()) 形式的全局名字空间。合起来后，我们可以得到以前者命名的类的引用。(回想一下，类是对象，并且它们可以像其它对象一样赋值给一个变量。) 如果 `parserName` 是字符串 `'ChefDialectizer'`，`parserClass` 将是类 `ChefDialectizer`。
[3]	最后，我们拥有了一个类对象 (`parserClass`)，接着我们想要生成这个类的一个实例。好，我们已经知道如何去做了：像函数一样调用类。这个类保存在一个局部变量中，但这个事实完全不会有什么影响；我们只是像函数一样调用这个局部变量，取出这个类的一个实例。如果 `parserClass` 是类 `ChefDialectizer`，`parser` 将是类 `ChefDialectizer` 的一个实例。

何必这么麻烦？毕竟只有三个 `Dialectizer` 类；为什么不只使用一个 `case` 语句？(噢，在 Python 中不存在 `case` 语句，但为什么不只使用一组 `if` 语句呢？) 理由之一是：可扩展性。这个 `translate` 函数完全不用关心我们定义了多少个方言变换器类。设想一下，如果我们明天定义了一个新的 `FooDialectizer` 类，把 `'foo'` 作为 `dialectName` 传给 `translate`，`translate` 也能工作。

甚至会更好。设想将 `FooDialectizer` 放进一个独立的模块中，使用 `from _module_ import` 将其导入。我们已经知道了，这样会将它包含在 `globals` () 中，所以不用修改 `translate`，它仍然可以正确运行，尽管 `FooDialectizer` 位于一个独立的文件中。

现在设想一下方言的名字是从程序外面的某个地方来的，也许是从一个数据库中，或从一个表格中的用户输入的值中。您可以使用任意多的服务端 Python 脚本架构来动态地生成网页；这个函数将接收在页面请求的查询字符串中的一个 URL 和一个方言名字 (两个都是字符串)，接着输出 “翻译” 后的网页。

最后，设想一下，使用了一种插件架构的 `Dialectizer` 框架。您可以将每个 `Dialectizer` 类放在分别放在独立的文件中，在 `dialect.py` 中只留下 `translate` 函数。假定一种统一的命名模式，这个 `translate` 函数能够动态地从合适的文件中导入合适的类，除了方言名字外什么都不用给出。(虽然您还没有看过动态导入，但我保证在后面的一章中会涉及到它。) 如果要加入一种新的方言，您只要在插件目录下加入一个以合适的名字命名的文件 (像 `foodialect.py`，它包含了 `FooDialectizer` 类)。使用方言名 `'foo'` 来调用这个 `translate` 函数，将会查找 `foodialect.py` 模块，导入 `FooDialectizer` 类，这样就行了。

例 8.22. `translate` 函数，第 3 部分

```
    parser.feed(htmlSource)
    parser.close()
    return parser.output()
```

[1]	剩下的工作似乎会非常无聊，但实际上，`feed` 函数执行了全部的转换工作。我们拥有存在于单个字符串中的全部 HTML 源代码，所以我们只需要调用 `feed` 一次。然而，您可以按您的需要经常调用 `feed`，分析器将不停地进行分析。所以如果我们担心内存的使用 (或者我们已经知道了将要处理非常巨大的 HTML 页面)，我们可以在一个循环中调用它，即我们读出一点 HTML 字节，就将其送进分析器。结果会是一样的。
[2]	因为 `feed` 维护着一个内部缓冲区，当您完成时，应该总是调用分析器的 `close` 方法 (那怕您像我们做的一样，一次就全部送出)。否则您可能会发现，输出丢掉了最后几个字节。
[3]	回想一下，`output` 是我们在 `BaseHTMLProcessor` 上定义的函数，用来将所有缓冲的输出片段连接起来并且以单个字符串返回。

像这样，我们已经 “翻译” 了一个网页，除了给出一个 URL 和一种方言的名字外，什么都没有给出。

进一步阅读

- 您可能会认为我的服务端脚本编程的想法是开玩笑。在我发现这个基于 web 的方言转换器之前，的确是这样想的。不幸的是，看不到它的源代码。

8.10. 小结

Python 向您提供了一个强大工具，`sgmllib.py`，可以通过将 HTML 结构转变为一种对象模型来进行处理。可以以许多不同的方式来使用这个工具。

- 对 HTML 进行分析，搜索特别的东西
- 摘录结果，如 URL lister
- 在处理过程中顺便调整结构，如给属性值加引号
- 将 HTML 转换为其它的东西，通过对文本进行处理，同时保留标记，如 `Dialectizer`

学过了这些例子之后，您应该无障碍地完成下面的事情：

- 使用 `locals` () 和 `globals` () 来访问名字空间
- 使用基于 dictionary 替换的字符串格式化

第 9 章 XML 处理

9.1. 概览

下面两章是关于 Python 中 XML 处理的。如果你已经对 XML 文档有了一个大概的了解，比如它是由结构化标记构成的，这些标记形成了层次模型的元素，等等这些知识都是有帮助的。如果你不明白这些，这里有很多 XML 教程能够解释这些基础知识。

如果你对 XML 不是很感兴趣，你还是应该读一下这些章节，它们涵盖了不少重要的主题，比如 Python 包、Unicode、命令行参数以及如何使用 `getattr` 进行方法分发。

如果你在大学里主修哲学 (而不是像计算机科学这样的实用专业)，并且曾不幸地被伊曼努尔·康德的著作折磨地够呛，那么你会非常欣赏本章的样例程序。(这当然不意味着你必须修过哲学。)

处理 XML 有两种基本的方式。一种叫做 SAX (“Simple API for XML”)，它的工作方式是，一次读出一点 XML 内容，然后对发现的每一个元素调用一个方法。(如果你读了 第 8 章 *HTML* 处理，这应该听起来很熟悉，因为这是 `sgmllib` 工作的方式。) 另一种方式叫做 DOM (“Document Object Model”)，它的工作方式是，一次性读入整个 XML 文档，然后使用 Python 类创建一个内部表示形式 (以树结构进行连接)。Python 拥有这两种解析方式的标准模块，但是本章只涉及 DOM。

下面是一个完整的 Python 程序，它根据 XML 格式定义的上下文无关语法生成伪随机输出。如果你不明白是什么意思，不用担心，下面两章中将会深入检视这个程序的输入和输出。

例 9.1. `kgp.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
"""Kant Generator for Python
Generates mock philosophy based on a context-free grammar
Usage: python kgp.py [options] [source]
Options:
  -g ..., --grammar=...   use specified grammar file or URL
  -h, --help              show this help
  -d                      show debugging information while parsing
Examples:
  kgp.py                  generates several paragraphs of Kantian philosophy
  kgp.py -g husserl.xml   generates several paragraphs of Husserl
  kpg.py "<xref id='paragraph'/>"  generates a paragraph of Kant
  kgp.py template.xml     reads from template.xml to decide what to generate
"""
from xml.dom import minidom
import random
import toolbox
import sys
import getopt
_debug = 0
class NoSourceError(Exception): pass
class KantGenerator:
    """generates mock philosophy based on a context-free grammar"""
    def __init__(self, grammar, source=None):
```

```
        self.loadGrammar(grammar)
        self.loadSource(source and source or self.getDefaultSource())
        self.refresh()
    def _load(self, source):
        """load XML input source, return parsed XML document
        - a URL of a remote XML file ("http://diveintopython.org/kant.xml")
        - a filename of a local XML file ("~/diveintopython/common/py/kant.xml")
        - standard input ("-")
        - the actual XML document, as a string
        """
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc
    def loadGrammar(self, grammar):
        """load context-free grammar"""
        self.grammar = self._load(grammar)
        self.refs = {}
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref
    def loadSource(self, source):
        """load source"""
        self.source = self._load(source)
    def getDefaultSource(self):
        """guess default source of the current grammar
        The default source will be one of the <ref>s that is not
        cross-referenced.  This sounds complicated but it's not.
        Example: The default source for kant.xml is
        "<xref id='section'/>", because 'section' is the one <ref>
        that is not <xref>'d anywhere in the grammar.
        In most grammars, the default source will produce the
        longest (and most interesting) output.
        """
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        if not standaloneXrefs:
            raise NoSourceError, "can't guess source, and no source specified"
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)
    def reset(self):
        """reset parser"""
        self.pieces = []
        self.capitalizeNextWord = 0
    def refresh(self):
        """reset output buffer, re-parse entire source file, and return output
        Since parsing involves a good deal of randomness, this is an
        easy way to get new output without having to reload a grammar file
        each time.
        """
        self.reset()
        self.parse(self.source)
        return self.output()
    def output(self):
        """output generated text"""
        return "".join(self.pieces)
    def randomChildElement(self, node):
        """choose a random child element of a node
        This is a utility method used by do_xref and do_choice.
        """
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE]
        chosen = random.choice(choices)
        if _debug:
            sys.stderr.write('%s available choices: %s\n' % \
                (len(choices), [e.toxml() for e in choices]))
            sys.stderr.write('Chosen: %s\n' % chosen.toxml())
        return chosen
    def parse(self, node):
        """parse a single XML node
        A parsed XML document (from minidom.parse) is a tree of nodes
```

```
        of various types.  Each node is represented by an instance of the
        corresponding Python class (Element for a tag, Text for
        text data, Document for the top-level document).  The following
        statement constructs the name of a class method based on the type
        of node we're parsing ("parse_Element" for an Element node,
        "parse_Text" for a Text node, etc.) and then calls the method.
        """
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__)
        parseMethod(node)
    def parse_Document(self, node):
        """parse the document node
        The document node by itself isn't interesting (to us), but
        its only child, node.documentElement, is: it's the root node
        of the grammar.
        """
        self.parse(node.documentElement)
    def parse_Text(self, node):
        """parse a text node
        The text of a text node is usually added to the output buffer
        verbatim.  The one exception is that <p class='sentence'> sets
        a flag to capitalize the first letter of the next word.  If
        that flag is set, we capitalize the text and reset the flag.
        """
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)
    def parse_Element(self, node):
        """parse an element
        An XML element corresponds to an actual tag in the source:
        <xref id='...'>, <p chance='...'>, <choice>, etc.
        Each element type is handled in its own method.  Like we did in
        parse(), we construct a method name based on the name of the
        element ("do_xref" for an <xref> tag, etc.) and
        call the method.
        """
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
    def parse_Comment(self, node):
        """parse a comment
        The grammar can contain XML comments, but we ignore them
        """
        pass
    def do_xref(self, node):
        """handle <xref id='...'> tag
        An <xref id='...'> tag is a cross-reference to a <ref id='...'>
        tag.  <xref id='sentence'/> evaluates to a randomly chosen child of
        <ref id='sentence'>.
        """
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
    def do_p(self, node):
        """handle <p> tag
        The <p> tag is the core of the grammar.  It can contain almost
        anything: freeform text, <choice> tags, <xref> tags, even other
        <p> tags.  If a "class='sentence'" attribute is found, a flag
        is set and the next word will be capitalized.  If a "chance='X'"
        attribute is found, there is an X% chance that the tag will be
        evaluated (and therefore a (100-X)% chance that it will be
        completely ignored)
        """
        keys = node.attributes.keys()
        if "class" in keys:
            if node.attributes["class"].value == "sentence":
                self.capitalizeNextWord = 1
        if "chance" in keys:
            chance = int(node.attributes["chance"].value)
            doit = (chance > random.randrange(100))
        else:
```

```
                doit = 1
            if doit:
                for child in node.childNodes: self.parse(child)
        def do_choice(self, node):
            """handle <choice> tag
            A <choice> tag contains one or more <p> tags.  One <p> tag
            is chosen at random and evaluated; the rest are ignored.
            """
            self.parse(self.randomChildElement(node))
def usage():
    print __doc__
def main(argv):
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg
    source = "".join(args)
    k = KantGenerator(grammar, source)
    print k.output()
if __name__ == "__main__":
    main(sys.argv[1:])
```

例 9.2. `toolbox.py`

```
"""Miscellaneous utility functions"""
def openAnything(source):
    """URI, filename, or string --> stream
    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner.  Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.
    Examples:
    >>> from xml.dom import minidom
    >>> sock = openAnything("http://localhost/kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("c:\\inetpub\\wwwroot\\kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("<ref id='conjunction'><text>and</text><text>or</text></ref>"
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    """
    if hasattr(source, "read"):
        return source
    if source == '-':
        import sys
        return sys.stdin
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)
    except (IOError, OSError):
        pass
    # try to open with native open function (if source is pathname)
    try:
        return open(source)
    except (IOError, OSError):
        pass
    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))
```

独立运行程序 `kgp.py`，它会解析 `kant.xml` 中默认的基于 XML 的语法，并以康德的风格打印出几段有哲学价值的段落来。

例 9.3. `kgp.py` 的样例输出

```
[you@localhost kgp]$ python kgp.py
 As is shown in the writings of Hume, our a priori concepts, in
reference to ends, abstract from all content of knowledge; in the study
of space, the discipline of human reason, in accordance with the
principles of philosophy, is the clue to the discovery of the
Transcendental Deduction.  The transcendental aesthetic, in all
theoretical sciences, occupies part of the sphere of human reason
concerning the existence of our ideas in general; still, the
never-ending regress in the series of empirical conditions constitutes
the whole content for the transcendental unity of apperception.  What
we have alone been able to show is that, even as this relates to the
architectonic of human reason, the Ideal may not contradict itself, but
it is still possible that it may be in contradictions with the
employment of the pure employment of our hypothetical judgements, but
natural causes (and I assert that this is the case) prove the validity
of the discipline of pure reason.  As we have already seen, time (and
it is obvious that this is true) proves the validity of time, and the
architectonic of human reason, in the full sense of these terms,
abstracts from all content of knowledge.  I assert, in the case of the
discipline of practical reason, that the Antinomies are just as
necessary as natural causes, since knowledge of the phenomena is a
posteriori.
    The discipline of human reason, as I have elsewhere shown, is by
its very nature contradictory, but our ideas exclude the possibility of
the Antinomies.  We can deduce that, on the contrary, the pure
employment of philosophy, on the contrary, is by its very nature
contradictory, but our sense perceptions are a representation of, in
the case of space, metaphysics.  The thing in itself is a
representation of philosophy.  Applied logic is the clue to the
discovery of natural causes.  However, what we have alone been able to
show is that our ideas, in other words, should only be used as a canon
for the Ideal, because of our necessary ignorance of the conditions.
[...snip...]
```

这当然是胡言乱语。噢，不完全是胡言乱语。它在句法和语法上都是正确的 (尽管非常罗嗦——康德可不是你们所说的踩得到点上的那种人)。其中一些实际上是正确的 (或者至少康德可能会认同的事情)，其中一些则明显是错误的，大部分只是语无伦次。但所有内容都符合康德的风格。

让我重复一遍，如果你现在或曾经主修哲学专业，这会非常、非常有趣。

有趣之处在于，这个程序中没有一点内容是属于康德的。所有的内容都来自于上下文无关语法文件 `kant.xml`。如果你要程序使用不同的语法文件 (可以在命令行中指定)，输出信息将完全不同。

例 9.4. `kgp.py` 的简单输出

```
[you@localhost kgp]$ python kgp.py -g binary.xml
00101001
[you@localhost kgp]$ python kgp.py -g binary.xml
10110100
```

在本章后面的内容中，你将近距离地观察语法文件的结构。现在，你只要知道语法文件定义了输出信息的结构，而 `kgp.py` 程序读取语法规则并随机确定哪些单词插入哪里。

9.2. 包

实际上解析一个 XML 文档是很简单的：只要一行代码。但是，在你接触那行代码前，需要暂时岔开一下，讨论一下包。

例 9.5. 载入一个 XML 文档 (偷瞥一下)

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
```

[1]	这个语法你之前没有见过。它看上去很像我们熟知的 `from _module_ import`，但是 `"."` 使得它好像不只是普通的 import 那么简单。事实上，`xml` 称为包，`dom` 是 `xml` 中嵌套的包，而 `minidom` 是 `xml.dom` 中的模块。

听起来挺复杂的，其实不是。看一下确切的实现可能会有帮助。包不过是模块的目录；嵌套包是子目录。一个包 (或一个嵌套包) 中的模块也只是 `.py` 文件罢了，永远都是，只是它们是在一个子目录中，而不是在你的 Python 安装环境的主 `lib/` 目录下。

例 9.6. 包的文件布局

```
Python21/           Python 安装根目录 (可执行文件的所在地)
|
+--lib/             库目录 (标准库模块的所在地)
   |
   +-- xml/         xml包 (实际上目录中还有其它东西)
       |
       +--sax/      xml.sax包 (也只是一个目录)
       |
       +--dom/      xml.dom包 (包含 minidom.py)
       |
       +--parsers/  xml.parsers包 (内部使用)
```

所以你说 `from xml.dom import minidom`，Python 认为它的意思是“在 `xml` 目录中查找 `dom` 目录，然后在这个目录 中查找 `minidom` 模块，接着导入它并以 `minidom` 命名 ”。但是 Python 更聪明；你不仅可以导入包含在一个包中的所有模块，还可以从包的模块中有选择地导入指定的类或者函数。语法都是一样的； Python 会根据包的布局理解你的意思，然后自动进行正确的导入。

例 9.7. 包也是模块

```
>>> from xml.dom import minidom
>>> minidom
<module 'xml.dom.minidom' from 'C:\Python21\lib\xml\dom\minidom.pyc'>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml.dom.minidom import Element
>>> Element
<class xml.dom.minidom.Element at 01095744>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml import dom
>>> dom
<module 'xml.dom' from 'C:\Python21\lib\xml\dom\__init__.pyc'>
>>> import xml
>>> xml
<module 'xml' from 'C:\Python21\lib\xml\__init__.pyc'>
```

[1]	这里你正从一个嵌套包 (`xml.dom`)中导入一个模块 (`minidom`)。结果就是 `minidom` 被导入到了你 (程序) 的命名空间中了。要引用 `minidom` 模块中的类 (比如 `Element`)，你必须在它们的类名前面加上模块名。
[2]	这里你正从一个来自嵌套包 (`xml.dom`) 的模块 (`minidom`) 中导入一个类 (`Element`)。结果就是 `Element` 直接导入到了你 (程序) 的命名空间中。注意，这样做并不会干扰以前的导入；现在 `Element` 类可以用两种方式引用了 (但其实是同一个类)。
[3]	这里你正在导入 `dom` 包 (`xml` 的一个嵌套包)，并将其作为一个模块。一个包的任何层次都可以视为一个模块，一会儿就会看到。它甚至可以拥有自己的属性和方法，就像你在前面看到过的模块。
[4]	这里你正在将根层次的 `xml` 包作为一个模块导入。

那么如何才能导入一个包 (它不过是磁盘上的一个目录) 并使其成为一个模块 (它总是在磁盘上的一个文件) 呢？答案就是神奇的 `__init__.py` 文件。你明白了吧，包不只是目录，它们是包含一个特殊文件 `__init__.py` 的目录。这个文件定义了包的属性和方法。例如，`xml.dom` 包含了 `Node` 类，它在 `xml/dom/__init__.py` 中有所定义。当你将一个包作为模块导入 (比如从 `xml` 导入 `dom`) 的时候，实际上导入了它的 `__init__.py` 文件。

> 注意
> 一个包是一个其中带有特殊文件 `__init__.py` 的目录。`__init__.py` 文件定义了包的属性和方法。其实它可以什么也不定义；可以只是一个空文件，但是必须要存在。如果 `__init__.py` 不存在，这个目录就仅仅是一个目录，而不是一个包，它就不能被导入或者包含其它的模块和嵌套包。

那为什么非得用包呢？嗯，它们提供了在逻辑上将相关模块归为一组的方法。不使用其中带有 `sax` 和 `dom` 的 `xml` 包，作者也可以选择将所有的 `sax` 功能放入 `xmlsax.py` 中，并将所有的 `dom` 功能放入 `xmldom.py` 中，或者干脆将所有东西放入单个模块中。但是这样可能不实用 (写到这儿时，XML 包已经超过了 3000 行代码) 并且很难管理 (独立的源文件意味着多个人可以同时在不同的地方进行开发)。

如果你发现自己正在用 Python 编写一个大型的子系统 (或者，很有可能，当你意识到你的小型子系统已经成长为一个大型子系统时)，你应该花费些时间设计一个好的包架构。它是 Python 所擅长的事情之一，所以应该好好利用它。

9.3. XML 解析

正如我说的，实际解析一个 XML 文档是非常简单的：只要一行代码。从这里出发到哪儿去就是你自己的事了。

例 9.8. 载入一个 XML 文档 (这次是真的)

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
>>> xmldoc
<xml.dom.minidom.Document instance at 010BE87C>
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

[1]	正如在上一节看到的，该语句从 `xml.dom` 包中导入 `minidom` 模块。
[2]	这就是进行所有工作的一行代码：`minidom.parse` 接收一个参数并返回 XML 文档解析后的表示形式。这个参数可以是很多东西；在本例中，它只是我本地磁盘上一个 XML 文档的文件名。(你需要将路径改为指向下载的例子所在的目录。) 但是你也可以传入一个文件对象，或甚至是一个类文件对象。这样你就可以在本章后面好好利用这一灵活性了。
[3]	从 `minidom.parse` 返回的对象是一个 `Document` 对象，它是 `Node` 类的一个子对象。这个 `Document` 对象是联锁的 Python 对象的一个复杂树状结构的根层次，这些 Python 对象完整表示了传给 `minidom.parse` 的 XML 文档。
[4]	`toxml` 是 `Node` 类的一个方法 (因此可以在从 `minidom.parse` 中得到的 `Document` 对象上使用)。`toxml` 打印出了 `Node` 表示的 XML。对于 `Document` 节点，这样就会打印出整个 XML 文档。

现在内存中已经有了一个 XML 文档了，你可以开始遍历它了。

例 9.9. 获取子节点

```
>>> xmldoc.childNodes
[<DOM Element: grammar at 17538908>]
>>> xmldoc.childNodes[0]
<DOM Element: grammar at 17538908>
>>> xmldoc.firstChild
<DOM Element: grammar at 17538908>
```

[1]	每个 `Node` 都有一个 `childNodes` 属性，它是一个 `Node` 对象的列表。一个 `Document` 只有一个子节点，即 XML 文档的根元素 (在本例中，是 `grammar` 元素)。
[2]	为了得到第一个 (在本例中，只有一个) 子节点，只要使用正规的列表语法。回想一下，其实这里没有发生什么特别的；这只是一个由正规 Python 对象构成的正规 Python 列表。
[3]	鉴于获取某个节点的第一个子节点是有用而且常见的行为，所以 `Node` 类有一个 `firstChild` 属性，它和 `childNodes[0]` 具有相同的语义。(还有一个 `lastChild` 属性，它和 `childNodes[-1]` 具有相同的语义。)

例 9.10. `toxml` 用于任何节点

```
>>> grammarNode = xmldoc.firstChild
>>> print grammarNode.toxml()
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

[1]	由于 `toxml` 方法是定义在 `Node` 类中的，所以对任何 XML 节点都是可用的，不仅仅是 `Document` 元素。

例 9.11. 子节点可以是文本

```
>>> grammarNode.childNodes
[<DOM Text node "\n">, <DOM Element: ref at 17533332>, \
<DOM Text node "\n">, <DOM Element: ref at 17549660>, <DOM Text node "\n">]
>>> print grammarNode.firstChild.toxml()

>>> print grammarNode.childNodes[1].toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print grammarNode.childNodes[3].toxml()
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
>>> print grammarNode.lastChild.toxml()
```

[1]	查看 `binary.xml` 中的 XML，你可能会认为 `grammar` 只有两个子节点，即两个 `ref` 元素。但是你忘记了一些东西：硬回车！在 `'<grammar>'` 之后，第一个 `'<ref>'` 之前是一个硬回车，并且这个文本算作 `grammar` 元素的一个子节点。类似地，在每个 `'</ref>'` 之后都有一个硬回车；它们都被当作子节点。所以 `grammar.childNodes` 实际上是一个有5个对象的列表：3个 `Text` 对象和两个 `Element` 对象。
[2]	第一个子节点是一个 `Text` 对象，它表示在 `'<grammar>'` 标记之后、第一个 `'<ref>'` 标记之后的硬回车。
[3]	第二个子节点是一个 `Element` 对象，表示了第一个 `ref` 元素。
[4]	第四个子节点是一个 `Element` 对象，表示了第二个 `ref` 元素。
[5]	最后一个子节点是一个 `Text` 对象，表示了在 `'</ref>'` 结束标记之后、 `'</grammar>'` 结束标记之前的硬回车。

例 9.12. 把文本挖出来

```
>>> grammarNode
<DOM Element: grammar at 19167148>
>>> refNode = grammarNode.childNodes[1]
>>> refNode
<DOM Element: ref at 17987740>
>>> refNode.childNodes
[<DOM Text node "\n">, <DOM Text node "  ">, <DOM Element: p at 19315844>, \
<DOM Text node "\n">, <DOM Text node "  ">, \
<DOM Element: p at 19462036>, <DOM Text node "\n">]
>>> pNode = refNode.childNodes[2]
>>> pNode
<DOM Element: p at 19315844>
>>> print pNode.toxml()
<p>0</p>
>>> pNode.firstChild
<DOM Text node "0">
>>> pNode.firstChild.data
u'0'
```

[1]	正如你在前面的例子中看到的，第一个 `ref` 元素是 `grammarNode.childNodes[1]`，因为 childNodes[0] 是一个代表硬回车的 `Text` 节点。
[2]	`ref` 元素有它自己的子节点集合，一个表示硬回车，另一个表示空格，一个表示 `p` 元素，诸如此类。
[3]	你甚至可以在这里使用 `toxml` 方法，尽管它深深嵌套在文档中。
[4]	`p` 元素只有一个子节点 (在这个例子中无法看出，但是如果你不信，可以看看 `pNode.childNodes`)，而且它是表示单字符 `'0'` 的一个 `Text` 节点。
[5]	`Text` 节点的 `.data` 属性可以向你提供文本节点真正代表的字符串。但是字符串前面的 `'u'` 是什么意思呢？答案将自己专门有一部分来论述。

9.4. Unicode

Unicode 是一个系统，用来表示世界上所有不同语言的字符。当 Python 解析一个 XML 文档时，所有的数据都是以unicode的形式保存在内存中的。

一会儿你就会了解，但首先，先看一些背景知识。

历史注解. 在 unicode 之前，对于每一种语言都存在独立的字符编码系统，每个系统都使用相同的数字(0-255)来表示这种语言的字符。一些语言 (像俄语) 对于如何表示相同的字符还有几种有冲突的标准；另一些语言 (像日语) 拥有太多的字符，需要多个字符集。在系统之间进行文档交流是困难的，因为对于一台计算机来说，没有方法可以识别出文档的作者使用了哪种编码模式；计算机看到的只是数字，并且这些数字可以表示不同的东西。接着考虑到试图将这些 (采用不同编码的) 文档存放到同一个地方 (比如在同一个数据库表中)；你需要在每段文本的旁边保存字符的编码，并且确保在传递文本的同时将编码也进行传递。接着考虑多语言文档，即在同一文档中使用了不同语言的字符。(比较有代表性的是使用转义符来进行模式切换；噗，我们处于俄语 koi8-r 模式，所以字符 241 表示这个；噗，现在我们处于 Mac 希腊语模式，所以字符 241 表示其它什么。等等。) 这些就是 unicode 被设计出来要解决的问题。

为了解决这些问题，unicode 用一个 2 字节数字表示每个字符，从 0 到 65535。[8] 每个 2 字节数字表示至少在一种世界语言中使用的一个唯一字符。(在多种语言中都使用的字符具有相同的数字码。) 这样就确保每个字符一个数字，并且每个数字一个字符。Unicode 数据永远不会模棱两可。

当然，仍然还存在着所有那些遗留的编码系统的情况。例如，7 位 ASCII，它可以将英文字符存诸为从 0 到 127 的数值。(65 是大写字母 “`A`”，97 是小写字母 “`a`”，等等。) 英语有着非常简单的字母表，所以它可以完全用 7 位 ASCII 来表示。像法语、西班牙语和德语之类的西欧语言都使用叫做 ISO-8859-1 的编码系统 (也叫做“latin-1”)，它使用 7 位 ASCII 字符表示从 0 到 127 的数字，但接着扩展到了 128-255 的范围来表示像 n 上带有一个波浪线(241)，和 u 上带有两个点(252)的字符。Unicode 在 0 到 127 上使用了同 7 位 ASCII 码一样的字符表，在 128 到 255上同 ISO-8859-1 一样，接着使用剩余的数字，256 到 65535，扩展到表示其它语言的字符。

在处理 unicode 数据时，在某些地方你可能需要将数据转换回这些遗留编码系统之一。例如，为了同其它一些计算机系统集成，这些系统期望它的数据使用一种特定的单字节编码模式，或将数据打印输出到一个不识别 unicode 的终端或打印机。或将数据保存到一个明确指定编码模式的 XML 文档中。

在了解这个注解之后，让我们回到 Python上来。

从 2.0 版开始，Python 整个语言都已经支持 unicode。XML 包使用 unicode 来保存所有解析了的 XML 数据，而且你可以在任何地方使用 unicode。

例 9.13. unicode 介绍

```
>>> s = u'Dive in'
>>> s
u'Dive in'
>>> print s
Dive in
```

[1]	为了创建一个 unicode 字符串而不是通常的 ASCII 字符串，要在字符串前面加上字母 “ `u` ”。注意这个特殊的字符串没有任何非 ASCII 的字符。这样很好；unicode 是 ASCII 的一个超集 (一个非常大的超集)，所以任何正常的 ASCII 都可以以 unicode 形式保存起来。
[2]	在打印字符串时，Python 试图将字符串转换为你的默认编码，通常是 ASCII 。(过会儿有更详细的说明。) 因为组成这个 unicode 字符串的字符都是 ASCII 字符，打印结果与打印正常的 ASCII 字符串是一样的；转换是无缝的，而且如果你没有注意到 `s` 是一个 unicode 字符串的话，你永远也不会注意到两者之间的差别。

例 9.14. 存储非 ASCII 字符

```
>>> s = u'La Pe\xf1a'
>>> print s
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> print s.encode('latin-1')
La Pe?a
```

[1]	unicode 真正的优势，理所当然的是它保存非 ASCII 字符的能力，例如西班牙语的 “ `?` ”(`n` 上带有一个波浪线)。用来表示波浪线 n 的 unicode 字符编码是十六进制的 `0xf1` (十进制的241)，你可以像这样输入： `\xf1` 。
[2]	还记得我说过 `print` 函数会尝试将 unicode 字符串转换为 ASCII 从而打印它吗？嗯，在这里将不会起作用，因为你的 unicode 字符串包含非 ASCII 字符，所以 Python 会引发 `UnicodeError` 异常。
[3]	这儿就是将 unicode 转换为其它编码模式起作用的地方。 `s` 是一个 unicode 字符串，但 `print` 只能打印正常的字符串。为了解决这个问题，我们调用 `encode` 方法 (它可以用于每个 unicode 字符串) 将 unicode 字符串转换为指定编码模式的正常字符串。我们向此函数传入一个参数。在本例中，我们使用 `latin-1` (也叫 `iso-8859-1`)，它包括带波浪线的 n (然而缺省的 ASCII 编码模式不包括，因为它只包含数值从 0 到 127 的字符)。

还记得我说过：需要从一个 unicode 得到一个正常字符串时，Python 通常默认将 unicode 转换成 ASCII 吗？嗯，这个默认编码模式是一个可以定制的选项。

例 9.15. `sitecustomize.py`

```
# sitecustomize.py
# this file can be anywhere in your Python path,
# but it usually goes in ${pythondir}/lib/site-packages/
import sys
sys.setdefaultencoding('iso-8859-1')
```

[1]	`sitecustomize.py` 是一个特殊的脚本；Python 会在启动的时候导入它，所以在其中的任何代码都将自动运行。就像注解中提到的那样，它可以放在任何地方 (只要 `import` 能够找到它)，但是通常它位于 Python 的 `lib` 目录的 `site-packages` 目录中。
[2]	嗯， `setdefaultencoding` 函数设置默认编码。Python 会在任何需要将 unicode 字符串自动转换为正规字符串的地方，使用这个编码模式。

例 9.16. 设置默认编码的效果

```
>>> import sys
>>> sys.getdefaultencoding()
'iso-8859-1'
>>> s = u'La Pe\xf1a'
>>> print s
La Pe?a
```

[1]	这个例子假设你已经按前一个例子中的改动对 `sitecustomize.py` 文件做了修改，并且已经重启了 Python。如果你的默认编码还是 `'ascii'`，可能你就没有正确设置 `sitecustomize.py` 文件，或者是没有重新启动 Python。默认的编码只能在 Python 启动的时候改变；之后就不能改变了。(由于一些我们现在不会仔细研究的古怪的编程技巧，你甚至不能在 Python 启动之后调用 `sys.setdefaultencoding` 函数。仔细研究 `site.py`，并搜索 “`setdefaultencoding`” 去发现为什么吧。)
[2]	现在默认的编码模式已经包含了你在字符串中使用的所有字符，Python 对字符串的自动强制转换和打印就不存在问题了。

例 9.17. 指定 `.py` 文件的编码

如果你打算在你的 Python 代码中保存非 ASCII 字符串，你需要在每个文件的顶端加入编码声明来指定每个 `.py` 文件的编码。这个声明定义了 `.py` 文件的编码为 UTF-8：

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
```

现在，想想 XML 中的编码应该是怎样的呢？不错，每一个 XML 文档都有指定的编码。重复一下，ISO-8859-1 是西欧语言存放数据的流行编码方式。KOI8-R 是俄语流行的编码方式。编码——如果指定了的话——都在 XML 文档的首部。

例 9.18. `russiansample.xml`

```
 <?xml version="1.0" encoding="koi8-r"?>  <preface>
<title>Предисловие</title>  </preface>
```

[1]	这是从一个真实的俄语 XML 文档中提取出来的示例；它就是这本书俄语翻译版的一部分。注意，编码 `koi8-r` 是在首部指定的。
[2]	这些是古代斯拉夫语的字符，就我所知，它们用来拼写俄语单词“Preface”。如果你在一个正常文本编辑器中打开这个文件，这些字符非常像乱码，因为它们使用了 `koi8-r` 编码模式进行编码，但是却以 `iso-8859-1` 编码模式进行显示。

例 9.19. 解析 `russiansample.xml`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('russiansample.xml')
>>> title = xmldoc.getElementsByTagName('title')[0].firstChild.data
>>> title
u'\u041f\u0440\u0435\u0434\u0438\u0441\u043b\u043e\u0432\u0438\u0435'
>>> print title
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> convertedtitle = title.encode('koi8-r')
>>> convertedtitle
'\xf0\xd2\xc5\xc4\xc9\xd3\xcc\xcf\xd7\xc9\xc5'
>>> print convertedtitle
Предисловие
```

[1]	我假设在这里你将前一个例子以 `russiansample.xml` 为名保存在当前目录中。也出于完整性的考虑，我假设你已经删除了 `sitecustomize.py` 文件，将缺省编码改回到 `'ascii'`，或至少将 `setdefaultencoding` 一行注释起来了。
[2]	注意 `title` 标记 (现在在 `title` 变量中，上面那一长串 Python 函数我们暂且跳过，下一节再解释)——在 XML 文档的 `title` 元素中的文本数据是以 unicode 保存的。
[3]	直接打印 title 是不可能的，因为这个 unicode 字符串包含了非 ASCII 字符，所以 Python 不能把它转换为 ASCII，因为它无法理解。
[4]	但是，你能够显式地将它转换为 `koi8-r`，在本例中，我们得到一个 (正常，非 unicode) 单字节字符的字符串 (`f0`，`d2`，`c5`，等等)，它是初始unicode字符串中字符 `koi8-r` 编码的版本。
[5]	打印 `koi8-r` 编码的字符串有可能会在你的屏幕上显示为乱码，因为你的 Python IDE 将这些字符作为 `iso-8859-1` 的编码进行解析，而不是 `koi8-r` 编码。但是，至少它们能打印。 (并且，如果你仔细看，当在一个不支持 unicode 的文本编辑器中打开最初的 XML 文档时，会看到相同的乱码。Python 在解析 XML 文档时，将它从 `koi8-r` 转换到了unicode，你只不过是将它转换回来。)

总结一下，如果你以前从没有看到过 unicode，倒是有些唬人，但是在 Python 处理 unicode 数据真是非常容易。如果你的 XML 文档都是 7 位的 ASCII (像本章中的例子)，你差不多永远都不用考虑 unicode。Python 在进行解析时会将 XML 文档中的 ASCII 数据转换为 unicode，在任何需要的时候强制转换回为 ASCII，你甚至永远都不用注意。但是如果你要处理其它语言的数据，Python 已经准备好了。

进一步阅读

- Unicode.org 是 unicode 标准的主页，包含了一个简要的技术简介。
- Unicode 教程有更多关于如何使用 Python unicode 函数的例子，包括甚至在并不真的需要时如何将 unicode 强制转换为 ASCII。
- PEP 263 涉及了何时、如何在你的 `.py` 文件中定义字符编码的更多细节。

Footnotes

[8] 这一点，很不幸仍然 过分简单了。现在 unicode 已经扩展用来处理古老的汉字、韩文和日文文本，它们有太多不同的字符，以至于 2 字节的 unicode 系统不能全部表示。但当前 Python 不支持超出范围的编码，并且我不知道是否有正在计划进行解决的项目。对不起，你已经到了我经验的极限了。

9.5. 搜索元素

通过一步步访问每一个节点的方式遍历 XML 文档可能很乏味。如果你正在寻找些特别的东西，又恰恰它们深深埋入了你的 XML 文档，有个捷径让你可以快速找到它：`getElementsByTagName` 。

在这部分，将使用 `binary.xml` 语法文件，它的内容如下：

例 9.20. `binary.xml`

```
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dt
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

它有两个 `ref` ， `'bit'` (位) 和 `'byte'` (字节)。一个 `bit` 是 `'0'` 或者 `'1'` ，而一个 `byte` 是 8 个 `bit` 。

例 9.21. `getElementsByTagName` 介绍

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref')
>>> reflist
[<DOM Element: ref at 136138108>, <DOM Element: ref at 136144292>]
>>> print reflist[0].toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print reflist[1].toxml()
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
```

[1]	`getElementsByTagName` 接收一个参数，即要找的元素的名称。它返回一个 `Element` 对象的列表，列表中的对象都是有指定名称的 XML 元素。在本例中，你能找到两个 `ref` 元素。

例 9.22. 每个元素都是可搜索的

```
>>> firstref = reflist[0]
>>> print firstref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> plist = firstref.getElementsByTagName("p")
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>]
>>> print plist[0].toxml()
<p>0</p>
>>> print plist[1].toxml()
<p>1</p>
```

[1]	继续前面的例子，在 `reflist` 中的第一个对象是 `'bit'` `ref` 元素。
[2]	你可以在这个 `Element` 上使用相同的 `getElementsByTagName` 方法来寻找所有在 `'bit'` `ref` 元素中的 `<p>` 元素。
[3]	和前面一样， `getElementsByTagName` 方法返回一个找到元素的列表。在本例中，你有两个元素，每“位”各占一个。

例 9.23. 搜索实际上是递归的

```
>>> plist = xmldoc.getElementsByTagName("p")
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>, <DOM Element: p at 1361461
>>> plist[0].toxml()
'<p>0</p>'
>>> plist[1].toxml()
'<p>1</p>'
>>> plist[2].toxml()
'<p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>'
```

[1]	仔细注意这个例子和前面例子之间的不同。前面，你是在 `firstref` 中搜索 `p` 元素，但是这里你是在 `xmldoc` 中搜索 `p` 元素， `xmldoc` 是代表了整个 XML 文档的根层对象。这样就会 找到嵌套在 `ref` 元素 (它嵌套在根 `grammar` 元素中) 中的 `p` 元素。
[2]	前两个 `p` 元素在第一个 `ref` 内 (`'bit'` `ref`)。
[3]	后一个 `p` 元素在第二个 `ref` 中 (`'byte'` `ref`)。

9.6. 访问元素属性

XML 元素可以有一个或者多个属性，只要已经解析了一个 XML 文档，访问它们就太简单了。

在这部分中，将使用 `binary.xml` 语法文件，你在上一节中已经看到过了。

> 注意
> 这部分由于某个含义重叠的术语可能让人有点糊涂。在一个 XML 文档中，元素可以有属性，而 Python 对象也有属性。当你解析一个 XML 文档时，你得到了一组 Python 对象，它们代表 XML 文档中的所有片段，同时有些 Python 对象代表 XML 元素的属性。但是表示 (XML) 属性的 (Python) 对象也有 (Python) 属性，它们用于访问对象表示的 (XML) 属性。我告诉过你它让人糊涂。我会公开提出关于如何更明显地区分这些不同的建议。

例 9.24. 访问元素属性

```
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref')
>>> bitref = reflist[0]
>>> print bitref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> bitref.attributes
<xml.dom.minidom.NamedNodeMap instance at 0x81e0c9c>
>>> bitref.attributes.keys()
[u'id']
>>> bitref.attributes.values()
[<xml.dom.minidom.Attr instance at 0x81d5044>]
>>> bitref.attributes["id"]
<xml.dom.minidom.Attr instance at 0x81d5044>
```

[1]	每个 `Element` 对象都有一个 `attributes` 属性，它是一个 `NamedNodeMap` 对象。听上去挺吓人的，其实不然，因为 `NamedNodeMap` 是一个行为像字典的对象，所以你已经知道怎么使用它了。
[2]	将 `NamedNodeMap` 视为一个字典，你可以通过 `attributes.keys()` 获得属性名称的一个列表。这个元素只有一个属性，`'id'` 。
[3]	属性名称，像其它 XML 文档中的文本一样，都是以 unicode 保存的。
[4]	再次将 `NamedNodeMap` 视为一个字典，你可以通过 `attributes.values()` 获取属性值的一个列表。这些值本身是 `Attr` 类型的对象。你将在下一个例子中看到如何获取对象的有用信息。
[5]	仍然把 `NamedNodeMap` 视为一个字典，你可以通过常用的字典语法和名称访问单个的属性。(那些非常认真的读者将已经知道 `NamedNodeMap` 类是如何实现这一技巧的：通过定义一个 `__getitem__` 专用方法。其它的读者可能乐意接受这一事实：他们不需要理解它是如何工作的就可以有效地使用它。)

例 9.25. 访问单个属性

```
>>> a = bitref.attributes["id"]
>>> a
<xml.dom.minidom.Attr instance at 0x81d5044>
>>> a.name
u'id'
>>> a.value
u'bit'
```

[1]	`Attr` 对象完整代表了单个 XML 元素的单个 XML 属性。属性的名称 (与你在 `bitref.attributes` `NamedNodeMap` 伪字典中寻找的对象同名) 保存在 `a.name` 中。
[2]	这个 XML 属性的真实文本值保存在 `a.value` 中。

> 注意
>
> 类似于字典，一个 XML 元素的属性没有顺序。属性可以以某种顺序偶然 列在最初的 XML 文档中，而在 XML 文档解析为 Python 对象时，`Attr` 对象以某种顺序偶然 列出，这些顺序都是任意的，没有任何特别的含义。你应该总是使用名称来访问单个属性，就像字典的键一样。

9.7. Segue [9]

以上就是 XML 的核心内容。下一章将使用相同的示例程序，但是焦点在于能使程序更加灵活的其它方面：使用输入流处理，使用 `getattr` 进行方法分发，并使用命令行标识允许用户重新配置程序而无需修改代码。

在进入下一章前，你应该没有困难的完成这些事情：

- 使用 `minidom` 解析 XML 文档，搜索已解析文档，并以任意顺序访问元素属性和元素子元素
- 将复杂的库组织为包
- 将 unicode 字符串转换为不同的字符编码

Footnotes

[9] “Segue”是音乐术语，意为“继续演奏”。

第 10 章 脚本和流

10.1. 抽象输入源

Python 的最强大力量之一是它的动态绑定，而动态绑定最强大的用法之一是类文件*(file-like)*对象。

许多需要输入源的函数可以只接收一个文件名，并以读方式打开文件，读取文件，处理完成后关闭它。其实它们不是这样的，而是接收一个类文件对象。

在最简单的例子中，类文件对象 是任意一个带有 `read` 方法的对象，这个方法带有一个可选的 `size` 参数，并返回一个字符串。调用时如果没有 `size` 参数，它从输入源中读取所有东西并将所有数据作为单个字符串返回；调用时如果指定了 `size` 参数，它将从输入源中读取 `size` 大小的数据并返回这些数据；再次调用的时候，它从余下的地方开始并返回下一块数据。

这就是从真实文件读取数据的工作方式；区别在于你不用把自己局限于真实的文件。输入源可以是任何东西：磁盘上的文件，甚至是一个硬编码的字符串。只要你将一个类文件对象传递给函数，函数只是调用对象的 `read` 方法，就可以处理任何类型的输入源，而不需要为处理每种类型分别编码。

你可能会纳闷，这和 XML 处理有什么关系。其实 `minidom.parse` 就是一个可以接收类文件对象的函数。

例 10.1. 从文件中解析 XML

```
>>> from xml.dom import minidom
>>> fsock = open('binary.xml')
>>> xmldoc = minidom.parse(fsock)
>>> fsock.close()
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

[1]	首先，你要打开一个磁盘上的文件。这会提供给你一个文件对象。
[2]	将文件对象传递给 `minidom.parse`，它调用 `fsock` 的 `read` 方法并从磁盘上的文件读取 XML 文档。
[3]	确保处理完文件后调用 `close` 方法。`minidom.parse` 不会替你做这件事。
[4]	在返回的 XML 文档上调用 `toxml()` 方法，打印出整个文档的内容。

哦，所有这些看上去像是在浪费大量的时间。毕竟，你已经看到，`minidom.parse` 可以只接收文件名，并自动执行所有打开文件和关闭无用文件的行为。不错，如果你知道正要解析的是一个本地文件，你可以传递文件名而且 `minidom.parse` 可以足够聪明地做正确的事情 (Do The Right Thing?[10])，这一切都不会有问题。但是请注意，使用类文件，会使分析直接从 Internet 上来的 XML 文档变得多么相似和容易！

例 10.2. 解析来自 URL 的 XML

```
>>> import urllib
>>> usock = urllib.urlopen('http://slashdot.org/slashdot.rdf')
>>> xmldoc = minidom.parse(usock)
>>> usock.close()
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<rdf:RDF xmlns="http://my.netscape.com/rdf/simple/0.9/"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<channel>
<title>Slashdot</title>
<link>http://slashdot.org/</link>
<description>News for nerds, stuff that matters</description>
</channel>
<image>
<title>Slashdot</title>
<url>http://images.slashdot.org/topics/topicslashdot.gif</url>
<link>http://slashdot.org/</link>
</image>
<item>
<title>To HDTV or Not to HDTV?</title>
<link>http://slashdot.org/article.pl?sid=01/12/28/0421241</link>
</item>
[...snip...]
```

[1]	正如在前一章中所看到的，`urlopen` 接收一个 web 页面的 URL 作为参数并返回一个类文件对象。最重要的是，这个对象有一个 `read` 方法，它可以返回 web 页面的 HTML 源代码。
[2]	现在把类文件对象传递给 `minidom.parse`，它顺从地调用对象的 `read` 方法并解析 `read` 方法返回的 XML 数据。这与 XML 数据现在直接来源于 web 页面的事实毫不相干。`minidom.parse` 并不知道 web 页面，它也不关心 web 页面；它只知道类文件对象。
[3]	到这里已经处理完毕了，确保将 `urlopen` 提供给你的类文件对象关闭。
[4]	顺便提一句，这个 URL 是真实的，它真的是一个 XML。它是 Slashdot 站点 (一个技术新闻和随笔站点) 上当前新闻提要的 XML 表示。

例 10.3. 解析字符串 XML (容易但不灵活的方式)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> xmldoc = minidom.parseString(contents)
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

[1]	`minidom` 有一个方法， `parseString` ，它接收一个字符串形式的完整 XML 文档作为参数并解析这个参数。如果你已经将整个 XML 文档放入一个字符串，你可以使用它代替 `minidom.parse` 。

好吧，所以你可以使用 `minidom.parse` 函数来解析本地文件和远端 URL，但对于解析字符串，你使用……另一个函数。这就是说，如果你要从文件、URL 或者字符串接收输入，就需要特别的逻辑来判断参数是否是字符串，然后调用 `parseString` 。多不让人满意。

如果有一个方法可以把字符串转换成类文件对象，那么你只要这个对象传递给 `minidom.parse` 就可以了。事实上，有一个模块专门设计用来做这件事： `StringIO` 。

例 10.4. `StringIO` 介绍

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> import StringIO
>>> ssock = StringIO.StringIO(contents)
>>> ssock.read()
"<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock.read()
''
>>> ssock.seek(0)
>>> ssock.read(15)
'<grammar><ref i'
>>> ssock.read(15)
"d='bit'><p>0</p"
>>> ssock.read()
'><p>1</p></ref></grammar>'
>>> ssock.close()
```

[1]	`StringIO` 模块只包含了一个类，也叫 `StringIO`，它允许你将一个字符串转换为一个类文件对象。`StringIO` 类在创建实例时接收字符串作为参数。
[2]	现在你有了一个类文件对象，你可用它做类文件的所有事情。比如 `read` 可以返回原始字符串。
[3]	再次调用 `read` 返回空字符串。真实文件对象的工作方式也是这样的；一旦你读取了整个文件，如果不显式定位到文件的开始位置，就不可能读取到任何其他数据。`StringIO` 对象以相同的方式进行工作。
[4]	使用 `StringIO` 对象的 `seek` 方法，你可以显式地定位到字符串的开始位置，就像在文件中定位一样。
[5]	将一个 `size` 参数传递给 `read` 方法，你还可以以块的形式读取字符串。
[6]	任何时候，`read` 都将返回字符串的未读部分。所有这些严格地按文件对象的方式工作；这就是术语类文件对象 的来历。

例 10.5. 解析字符串 XML (类文件对象方式)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock = StringIO.StringIO(contents)
>>> xmldoc = minidom.parse(ssock)
>>> ssock.close()
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

[1]	现在你可以把类文件对象 (实际是一个 `StringIO`) 传递给 `minidom.parse`，它将调用对象的 `read` 方法并高兴地开始解析，绝不会知道它的输入源自一个硬编码的字符串。

那么现在你知道了如何使用同一个函数，`minidom.parse`，来解析一个保存在 web 页面上、本地文件中或硬编码字符串中的 XML 文档。对于一个 web 页面，使用 `urlopen` 得到类文件对象；对于本地文件，使用 `open`；对于字符串，使用 `StringIO`。现在让我们进一步并归纳一下这些 不同。

例 10.6. `openAnything`

```
def openAnything(source):
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)
    except (IOError, OSError):
        pass
    # try to open with native open function (if source is pathname)
    try:
        return open(source)
    except (IOError, OSError):
        pass
    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))
```

[1]	`openAnything` 函数接受单个参数， `source` ，并返回类文件对象。 `source` 是某种类型的字符串；它可能是一个 URL (例如 `'http://slashdot.org/slashdot.rdf'`)，一个本地文件的完整或者部分路径名 (例如 `'binary.xml'`)，或者是一个包含了待解析 XML 数据的字符串。
[2]	首先，检查 `source` 是否是一个 URL。这里通过强制方式进行：尝试把它当作一个 URL 打开并静静地忽略打开非 URL 引起的错误。这样做非常好，因为如果 `urllib` 将来支持更多的 URL 类型，不用重新编码就可以支持它们。如果 `urllib` 能够打开 `source` ，那么 `return` 可以立刻把你踢出函数，下面的 `try` 语句将不会执行。
[3]	另一方面，如果 `urllib` 向你呼喊并告诉你 `source` 不是一个有效的 URL，你假设它是一个磁盘文件的路径并尝试打开它。再一次，你不用做任何特别的事来检查 `source` 是否是一个有效的文件名 (在不同的平台上，判断文件名有效性的规则变化很大，因此不管怎样做都可能会判断错)。反而，只要盲目地打开文件并静静地捕获任何错误就可以了。
[4]	到这里，你需要假设 `source` 是一个其中有硬编码数据的字符串 (因为没有别的可以判断的了)，所以你可以使用 `StringIO` 从中创建一个类文件对象并将它返回。(实际上，由于使用了 `str` 函数，所以 `source` 没有必要一定是字符串；它可以是任何对象，你可以使用它的字符串表示形式，只要定义了它的 `__str__` 专用方法。)

现在你可以使用这个 `openAnything` 函数联合 `minidom.parse` 构造一个函数，接收一个指向 XML 文档的 `source`，而且无需知道这个 `source` 的含义 (可以是一个 URL 或是一个本地文件名，或是一个硬编码 XML 文档的字符串形式)，然后解析它。

例 10.7. 在 `kgp.py` 中使用 `openAnything`

```
class KantGenerator:
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc
```

Footnotes

[10] 这是一部著名的电影。——译注

10.2. 标准输入、输出和错误

UNIX 用户已经对标准输入、标准输出和标准错误的概念非常熟悉了。这一节是为其他不熟悉的人准备的。

标准输入和标准错误 (通常缩写为 `stdout` 和 `stderr`) 是内建在每一个 UNIX 系统中的管道。当你 `print` 某些东西时，结果前往 `stdout` 管道；当你的程序崩溃并打印出调试信息 (例如 Python 中的 traceback (错误跟踪)) 的时候，信息前往 `stderr` 管道。通常这两个管道只与你正在工作的终端窗口相联，所以当一个程序打印时，你可以看到输出，而当一个程序崩溃时，你可以看到调试信息。(如果你正在一个基于窗口的 Python IDE 上工作，`stdout` 和 `stderr` 缺省为你的“交互窗口”。)

例 10.8. `stdout` 和 `stderr` 介绍

```
>>> for i in range(3):
... print 'Dive in'
Dive in
Dive in
Dive in
>>> import sys
>>> for i in range(3):
... sys.stdout.write('Dive in')
Dive inDive inDive in
>>> for i in range(3):
... sys.stderr.write('Dive in')
Dive inDive inDive in
```

[1]	正如在例 6.9 “简单计数”中看到的，你可以使用 Python 内置的 `range` 函数来构造简单的计数循环，即重复某物一定的次数。
[2]	`stdout` 是一个类文件对象；调用它的 `write` 函数可以打印出你给定的任何字符串。实际上，这就是 `print` 函数真正做的事情；它在你打印的字符串后面加上一个硬回车，然后调用 `sys.stdout.write` 函数。
[3]	在最简单的例子中，`stdout` 和 `stderr` 把它们的输出发送到相同的地方：Python IDE (如果你在一个 IDE 中的话)，或者终端 (如果你从命令行运行 Python 的话)。和 `stdout` 一样，`stderr` 并不为你添加硬回车；如果需要，要自己加上。

`stdout` 和 `stderr` 都是类文件对象，就像在第 10.1 节 “抽象输入源”中讨论的一样，但是它们都是只写的。它们都没有 `read` 方法，只有 `write` 方法。然而，它们仍然是类文件对象，因此你可以将其它任何 (类) 文件对象赋值给它们来重定向其输出。

例 10.9. 重定向输出

```
[you@localhost kgp]$ python stdout.py
Dive in
[you@localhost kgp]$ cat out.log
This message will be logged instead of displayed
```

(在 Windows 上，你要使用 `type` 来代替 `cat` 显示文件的内容。)

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
#stdout.py
import sys
print 'Dive in'
saveout = sys.stdout
fsock = open('out.log', 'w')
sys.stdout = fsock
print 'This message will be log
sys.stdout = saveout
fsock.close()
```

[1]	打印输出到 IDE “交互窗口” (或终端，如果从命令行运行脚本的话)。
[2]	始终在重定向前保存 `stdout` ，这样的话之后你还可以将其设回正常。
[3]	打开一个新文件用于写入。如果文件不存在，将会被创建。如果文件存在，将被覆盖。
[4]	所有后续的输出都会被重定向到刚才打开的新文件上。
[5]	这样只会将输出结果“打印”到日志文件中；在 IDE 窗口中或在屏幕上不会看到输出结果。
[6]	在我们将 `stdout` 搞乱之前，让我们把它设回原来的方式。
[7]	关闭日志文件。

重定向 `stderr` 以完全相同的方式进行，只要把 `sys.stdout` 改为 `sys.stderr` 。

例 10.10. 重定向错误信息

```
[you@localhost kgp]$ python stderr.py
[you@localhost kgp]$ cat error.log
Traceback (most recent line last):
  File "stderr.py", line 5, in ?
    raise Exception, 'this error will be logged'
Exception: this error will be logged
```

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
#stderr.py
import sys
fsock = open('error.log', 'w')
sys.stderr = fsock                          raise Exception, 'this error will be logged
```

[1]	打开你要存储调试信息的日志文件。
[2]	将新打开的日志文件的文件对象赋值给 `stderr` 以重定向标准错误。
[3]	引发一个异常。从屏幕输出上可以注意到这个行为没有 在屏幕上打印出任何东西。所有正常的跟踪信息已经写进 `error.log` 。
[4]	还要注意你既没有显式关闭日志文件，也没有将 `stderr` 设回最初的值。这样挺好，因为一旦程序崩溃 (由于引发的异常)，Python 将替我们清理并关闭文件，因此永远不恢复 `stderr` 不会造成什么影响。然而对于 `stdout` ，恢复初始值相对更为重要——你可能会在后面再次操作标准输出。

向标准错误写入错误信息是很常见的，所以有一种较快的语法可以立刻导出信息。

例 10.11. 打印到 `stderr`

```
>>> print 'entering function'
entering function
>>> import sys
>>> print >> sys.stderr, 'entering function'
entering function
```

[1]	`print` 语句的快捷语法可以用于写入任何打开的文件 (或者是类文件对象)。在这里，你可以将单个 `print` 语句重定向到 `stderr` 而且不用影响后面的 `print` 语句。

另一方面，标准输入是一个只读文件对象，它表示从前一个程序到这个程序的数据流。这个对于老的 Mac OS 用户和 Windows 用户可能不太容易理解，除非你受到过 MS-DOS 命令行的影响。在 MS-DOS 命令行中，你可以使用一行指令构造一个命令的链，使得一个程序的输出就可以成为下一个程序的输入。第一个程序只是简单地输出到标准输出上 (程序本身没有做任何特别的重定向，只是执行了普通的 `print` 语句等)，然后，下一个程序从标准输入中读取，操作系统就把一个程序的输出连接到一个程序的输入。

例 10.12. 链接命令

```
[you@localhost kgp]$ python kgp.py -g binary.xml
01100111
[you@localhost kgp]$ cat binary.xml
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dt
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
[you@localhost kgp]$ cat binary.xml | python kgp.py -g -
10110001
```

[1]	正如你在第 9.1 节 “概览”中看到的，该命令将只打印一个随机的八位字符串，其中只有 `0` 或者 `1` 。
[2]	该处只是简单地打印出整个 `binary.xml` 文档的内容。(Windows 用户应该用 `type` 代替 `cat` 。)
[3]	该处打印 `binary.xml` 的内容，但是“ `|` ”字符，称为“管道”符，说明内容不会打印到屏幕上；它们会成为下一个命令的标准输入，在这个例子中是你调用的 Python 脚本。
[4]	为了不用指定一个文件 (例如 `binary.xml`)，你需要指定“ `-` ”，它会使得你的脚本从标准输入载入脚本，而不是从磁盘上的特定文件。 (下一个例子更多地说明了这是如何实现的)。所以效果和第一种语法是一样的，在那里你要直接指定语法文件，但是想想这里的扩展性。让我们把 `cat binary.xml` 换成别的什么东西——例如运行一个脚本动态生成语法——然后通过管道将它导入你的脚本。它可以来源于任何地方：数据库，或者是生成语法的元脚本，或者其他。你根本不需要修改你的 `kgp.py` 脚本就可以并入这个功能。你要做的仅仅是从标准输入取得一个语法文件，然后你就可以将其他的逻辑分离出来，放到另一程序中去了。

那么脚本是如何“知道”在语法文件是“ `-` ”时从标准输入读取? 其实不神奇；它只是代码。

例 10.13. 在 `kgp.py` 中从标准输入读取

```
 def openAnything(source):
    if source == "-":
        import sys
        return sys.stdin
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
[... snip ...]
```

[1]	这是 `toolbox.py` 中的 `openAnything` 函数，以前在第 10.1 节 “抽象输入源”中你已经检视过了。所有你要做的就是在函数的开始加入 3 行代码来检测源是否是“ `-` ”；如果是，返回 `sys.stdin` 。就这么简单！记住， `stdin` 是一个拥有 `read` 方法的类文件对象，所以其它的代码 (在 `kgp.py` 中，在那里你调用了 `openAnything`) 一点都不需要改动。

10.3. 查询缓冲节点

`kgp.py` 使用了多种技巧，在你进行 XML 处理时，它们或许能派上用场。第一个就是，利用输入文档的结构稳定特征来构建节点缓冲。

一个语法文件定义了一系列的 `ref` 元素。每个 `ref` 包含了一个或多个 `p` 元素，`p` 元素则可以包含很多不同的东西，包括 `xref`。对于每个 `xref`，你都能找到相对应的 `ref` 元素 (它们具有相同的 `id` 属性)，然后选择 `ref` 元素的子元素之一进行解析。(在下一部分中你将看到是如何进行这种随机选择的。)

语法的构建方式如下：先为最小的片段定义 `ref` 元素，然后使用 `xref` 定义“包含”第一个 `ref` 元素的 `ref` 元素，等等。然后，解析“最大的”引用并跟着 `xref` 跳来跳去，最后输出真实的文本。输出的文本依赖于你每次填充 `xref` 时所做的 (随机) 决策，所以每次的输出都是不同的。

这种方式非常灵活，但是有一个不好的地方：性能。当你找到一个 `xref` 并需要找到相应的 `ref` 元素时，会遇到一个问题。`xref` 有 `id` 属性，而你要找拥有相同 `id` 属性的 `ref` 元素，但是没有简单的方式做到这件事。较慢的方式是每次获取所有 `ref` 元素的完整列表，然后手动遍历并检视每一个 `id` 属性。较快的方式是只做一次，然后以字典形式构建一个缓冲。

例 10.14. `loadGrammar`

```
    def loadGrammar(self, grammar):
        self.grammar = self._load(grammar)
        self.refs = {}
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref
```

[1]	从创建一个空字典 `self.refs` 开始。
[2]	正如你在第 9.5 节 “搜索元素”中看到的，`getElementsByTagName` 返回所有特定名称元素的一个列表。你可以很容易地得到所有 `ref` 元素的一个列表，然后遍历这个列表。
[3]	正如你在第 9.6 节 “访问元素属性”中看到的，使用标准的字典语法，你可以通过名称来访问个别元素。所以，`self.refs` 字典的键将是每个 `ref` 元素的 `id` 属性值。
[4]	`self.refs` 字典的值将是 `ref` 元素本身。如你在第 9.3 节 “XML 解析”中看到的，已解析 XML 文档中的每个元素、节点、注释和文本片段都是一个对象。

只要构建了这个缓冲，无论何时你遇到一个 `xref` 并且需要找到具有相同 `id` 属性的 `ref` 元素，都只需在 `self.refs` 中查找它。

例 10.15. 使用 `ref` 元素缓冲

```
def do_xref(self, node):
    id = node.attributes["id"].value
    self.parse(self.randomChildElement(self.refs[id]))
```

你将在下一部分探究 `randomChildElement` 函数。

10.4. 查找节点的直接子节点

解析 XML 文档时，另一个有用的己技巧是查找某个特定元素的所有直接子元素。例如，在语法文件中，一个 `ref` 元素可以有数个 `p` 元素，其中每一个都可以包含很多东西，包括其他的 `p` 元素。你只要查找作为 `ref` 孩子的 `p` 元素，不用查找其他 `p` 元素的孩子 `p` 元素。

你可能认为你只要简单地使用 `getElementsByTagName` 来实现这点就可以了，但是你不可以这么做。 `getElementsByTagName` 递归搜索并返回所有找到的元素的单个列表。由于 `p` 元素可以包含其他的 `p` 元素，你不能使用 `getElementsByTagName` ，因为它会返回你不要的嵌套 `p` 元素。为了只找到直接子元素，你要自己进行处理。

例 10.16. 查找直接子元素

```
    def randomChildElement(self, node):
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE]
        chosen = random.choice(choices)
        return chosen
```

[1]	正如你在例 9.9 “获取子节点”中看到的， `childNodes` 属性返回元素所有子节点的一个列表。
[2]	然而，正如你在例 9.11 “子节点可以是文本”中看到的， `childNodes` 返回的列表包含了所有不同类型的节点，包括文本节点。这并不是你在这里要查找的。你只要元素形式的孩子。
[3]	每个节点都有一个 `nodeType` 属性，它可以是 `ELEMENT_NODE` ， `TEXT_NODE` ， `COMMENT_NODE` ，或者其它值。可能值的完整列表在 `xml.dom` 包的 `__init__.py` 文件中。(关于包的介绍，参见第 9.2 节 “包”。) 但你只是对元素节点有兴趣，所以你可以过滤出一个列表，其中只包含 `nodeType` 是 `ELEMENT_NODE` 的节点。
[4]	只要拥有了一个真实元素的列表，选择任意一个都很容易。Python 有一个叫 `random` 的模块，它包含了好几个有用的函数。 `random.choice` 函数接收一个任意数量条目的列表并随机返回其中的一个条目。比如，如果 `ref` 元素包含了多个 `p` 元素，那么 `choices` 将会是 `p` 元素的一个列表，而 `chosen` 将被赋予其中的某一个值，而这个值是随机选择的。

10.5. 根据节点类型创建不同的处理器

第三个有用的 XML 处理技巧是将你的代码基于节点类型和元素名称分散到逻辑函数中。解析后的 XML 文档是由各种类型的节点组成的，每一个都是通过 Python 对象表示的。文档本身的根层次通过一个 `Document` 对象表示。`Document` 还包含了一个或多个 `Element` 对象 (表示 XML 标记)，其中的每一个可以包含其它的 `Element` 对象、`Text` 对象 (表示文本)，或者 `Comment` 对象 (表示内嵌注释)。使用 Python 编写分离各个节点类型逻辑的分发器非常容易。

例 10.17. 已解析 XML 对象的类名

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('kant.xml')
>>> xmldoc
<xml.dom.minidom.Document instance at 0x01359DE8>
>>> xmldoc.__class__
<class xml.dom.minidom.Document at 0x01105D40>
>>> xmldoc.__class__.__name__
'Document'
```

[1]	暂时假设 `kant.xml` 在当前目录中。
[2]	正如你在第 9.2 节 “包”中看到的，解析 XML 文档返回的对象是一个 `Document` 对象，就像在 `xml.dom` 包的 `minidom.py` 中定义的一样。又如你在第 5.4 节 “类的实例化”中看到的，`__class__` 是每个 Python 对象的一个内置属性。
[3]	此外，`__name__` 是每个 Python 类的内置属性，是一个字符串。这个字符串并不神秘；它和你在定义类时输入的类名相同。(参见第 5.3 节 “类的定义”。)

好，现在你能够得到任何给定 XML 节点的类名了 (因为每个 XML 节点都是以一个 Python 对象表示的)。你怎样才能利用这点来分离解析每个节点类型的逻辑呢？答案就是 `getattr`，你第一次见它是在第 4.4 节 “通过 getattr 获取对象引用”中。

例 10.18. `parse`，通用 XML 节点分发器

```
    def parse(self, node):
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__)
        parseMethod(node)
```

[1]	首先，注意你正在基于传入节点 (`node` 参数) 的类名构造一个较大的字符串。所以如果你传入一个 `Document` 节点，你就构造了字符串 `'parse_Document'`，其它类同于此。
[2]	现在你可以把这个字符串当作一个函数名称，然后通过 `getattr` 得到函数自身的引用。
[3]	最后，你可以调用函数并将节点自身作为参数传入。下一个例子将展示每个函数的定义。

例 10.19. `parse` 分发器调用的函数

```
def parse_Document(self, node):
    self.parse(node.documentElement)
def parse_Text(self, node):
    text = node.data
    if self.capitalizeNextWord:
        self.pieces.append(text[0].upper())
        self.pieces.append(text[1:])
        self.capitalizeNextWord = 0
    else:
        self.pieces.append(text)
def parse_Comment(self, node):
    pass
def parse_Element(self, node):
    handlerMethod = getattr(self, "do_%s" % node.tagName)
    handlerMethod(node)
```

[1]	`parse_Document` 只会被调用一次，因为在一个 XML 文档中只有一个 `Document` 节点，并且在已解析 XML 的表示中只有一个 `Document` 对象。在此它只是起到中转作用，转而解析语法文件的根元素。
[2]	`parse_Text` 在节点表示文本时被调用。这个函数本身做某种特殊处理，自动将句子的第一个单词进行大写处理，而不是简单地将表示的文本追加到一个列表中。
[3]	`parse_Comment` 只有一个 `pass`，因为你并不关心语法文件中嵌入的注释。但是注意，你还是要定义这个函数并显式地让它不做任何事情。如果这个函数不存在，通用 `parse` 函数在遇到一个注释的时候会执行失败，因为它试图找到并不存在的 `parse_Comment` 函数。为每个节点类型定义独立的函数——甚至你不要使用的——将会使通用 `parse` 函数保持简单和沉默。
[4]	`parse_Element` 方法其实本身就是一个分发器，一个基于元素的标记名称的分发器。这个基本概念是相同的：使用元素的区别 (它们的标记名称) 然后针对每一个分发到一个独立的函数。你构建了一个类似于 `'do_xref'` 的字符串 (对 `<xref>` 标记而言)，找到这个名称的函数，并调用它。对其它的标记名称 (像 `<p>` 和 `<choice>`) 在解析语法文件的时候都可以找到类似的函数。

在这个例子中，分发函数 `parse` 和 `parse_Element` 只是找到相同类中的其它方法。如果你进行的处理过程很复杂 (或者你有很多不同的标记名称)，你可以将代码分散到独立的模块中，然后使用动态导入的方式导入每个模块并调用你需要的任何函数。动态导入将在第 16 章 函数编程中介绍。

10.6. 处理命令行参数

Python 完全支持创建在命令行运行的程序，也支持通过命令行参数和短长样式来指定各种选项。这些并非是 XML 特定的，但是这样的脚本可以充分使用命令行处理，看来是时候提一下它了。

如果不理解命令行参数如何暴露给你的 Python 程序，讨论命令行处理是很困难的，所以让我们先写个简单点的程序来看一下。

例 10.20. `sys.argv` 介绍

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
#argecho.py
import sys
for arg in sys.argv:
    print arg
```

[1]	每个传递给程序的命令行参数都在 `sys.argv` 中，而它仅仅是一个列表。这里我们在独立行中打印出每个参数。

例 10.21. `sys.argv` 的内容

```
[you@localhost py]$ python argecho.py
argecho.py
[you@localhost py]$ python argecho.py abc def
argecho.py
abc
def
[you@localhost py]$ python argecho.py --help
argecho.py
--help
[you@localhost py]$ python argecho.py -m kant.xml
argecho.py
-m
kant.xml
```

[1]	关于 `sys.argv` 需要了解的第一件事情就是：它包含了你正在调用的脚本的名称。你后面会实际使用这个知识，在第 16 章 函数编程中。现在不用担心。
[2]	命令行参数通过空格进行分隔。在 `sys.argv` 列表中，每个参数都是一个独立的元素。
[3]	命令行标志，像 `--help`，在 `sys.argv` 列表中还保存了它们自己的元素。
[4]	为了让事情更有趣，有些命令行标志本身就接收参数。比如，这里有一个标记 (`-m`) 接收一个参数 (`kant.xml`)。标记自身和标记参数只是 `sys.argv` 列表中的一串元素。并没有试图将元素与其它元素进行关联；所有你得到的是一个列表。

所以正如你所看到的，你确实拥有了命令行传入的所有信息，但是接下来要实际使用它似乎不那么容易。对于只是接收单个参数或者没有标记的简单程序，你可以简单地使用 `sys.argv[1]` 来访问参数。这没有什么羞耻的；我一直都是这样做的。对更复杂的程序，你需要 `getopt` 模块。

例 10.22. `getopt` 介绍

```
 def main(argv):
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
...
if __name__ == "__main__":
    main(sys.argv[1:])
```

[1]	首先，看一下例子的最后并注意你正在调用 `main` 函数，参数是 `sys.argv[1:]`。记住，`sys.argv[0]` 是你正在运行脚本的名称；在处理命令行时，你不用关心它，所以你可以砍掉它并传入列表的剩余部分。
[2]	这里就是所有有趣处理发生的地方。`getopt` 模块的 `getopt` 函数接受三个参数：参数列表 (你从 `sys.argv[1:]` 得到的)、一个包含了程序所有可能接收到的单字符命令行标志，和一个等价于单字符的长命令行标志的列表。第一次看的时候，这有点混乱，下面有更多的细节解释。
[3]	在解析这些命令行标志时，如果有任何事情错了，`getopt` 会抛出异常，你可以捕获它。你可以告诉 `getopt` 你明白的所有标志，那么这也意味着终端用户可以传入一些你不理解的命令行标志。
[4]	和 UNIX 世界中的标准实践一样，如果脚本被传入了不能理解的标志，你要打印出正确用法的一个概要并友好地退出。注意，在这里我没有写出 `usage` 函数。你还是要在某个地方写一个，使它打印出合适的概要；它不是自动的。

那么你传给 `getopt` 函数的参数是什么呢？好的，第一个只不过是一个命令行标志和参数的原始列表 (不包括第一个元素——脚本名称，你在调用 `main` 函数之前就已经将它砍掉了)。第二个是脚本接收的短命令行标志的一个列表。

`"hg:d"`

`-h`

打印用法概要

`-g ...`

使用给定的语法文件或 URL

`-d`

在解析时显示调试信息

第一个标志和第三个标志是简单的独立标志；你选择是否指定它们，它们做某些事情 (打印帮助) 或者改变状态 (打开调试)。但是，第二个标志 (`-g`) 必须 跟随一个参数——进行读取的语法文件的名称。实际上，它可以是一个文件名或者一个 web 地址，这时还不知道 (后面会确定)，但是你要知道必须要有些东西。所以，你可以通过在 `getopt` 函数的第二个参数的 `g` 后面放一个冒号，来向 `getopt` 说明这一点。

更复杂的是，这个脚本既接收短标志 (像 `-h`)，也接受长标志 (像 `--help`)，并且你要它们做相同的事。这就是 `getopt` 第三个参数存在的原因：它是指定长标志的一个列表，其中的长标志是和第二个参数中指定的短标志相对应的。

`["help", "grammar="]`

`--help`

打印用法概要

`--grammar ...`

使用给定的语法文件或 URL

这里有三点要注意：

1. 所有命令行中的长标志以两个短划线开始，但是在调用 `getopt` 时，你不用包含这两个短划线。它们是能够被理解的。
2. `--grammar` 标志的后面必须跟着另一个参数，就像 `-g` 标志一样。通过等于号标识出来： `"grammar="` 。
3. 长标志列表比短标志列表更短一些，因为 `-d` 标志没有相应的长标志。这很好；只有 `-d` 才会打开调试。但是短标志和长标志的顺序必须是相同的，你应该先指定有长标志的短标志，然后才是剩下的短标志。

被搞昏没？让我们看一下真实的代码，看看它在上下文中是否起作用。

例 10.23. 在 `kgp.py` 中处理命令行参数

```
def main(argv):
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg
    source = "".join(args)
    k = KantGenerator(grammar, source)
    print k.output()
```

[1]	`grammar` 变量会跟踪你正在使用的语法文件。如果你没有在命令行指定它 (使用 `-g` 或者 `--grammar` 标志定义它)，在这里你将初始化它。
[2]	你从 `getopt` 取回的 `opts` 变量是一个由元组 (`flag` 和 `argument`) 组成的列表。如果标志没有带任何参数，那么 `arg` 只是 `None` 。这使得遍历标志更容易了。
[3]	`getopt` 验证命令行标志是否可接受，但是它不会在短标志和长标志之间做任何转换。如果你指定 `-h` 标志， `opt` 将会包含 `"-h"` ；如果你指定 `--help` 标志， `opt` 将会包含 `"--help"` 标志。所以你需要检查它们两个。
[4]	别忘了， `-d` 标志没有相应的长标志，所以你只需要检查短形式。如果你找到了它，你就可以设置一个全局变量来指示后面要打印出调试信息。(我习惯在脚本的开发过程中使用它。什么，你以为所有这些程序都是一次成功的？)
[5]	如果你找到了一个语法文件，跟在 `-g` 或者 `--grammar` 标志后面，那你就要把跟在后面的参数 (`arg`) 保存到变量 `grammar` 中，覆盖掉在 `main` 函数你初始化的默认值。
[6]	就是这样。你已经遍历并处理了所有的命令行标志。这意味着所有剩下的东西都必须是命令行参数。它们由 `getopt` 函数的 `args` 变量返回。在这个例子中，你把它们当作了解析器源材料。如果没有指定命令行参数， `args` 将是一个空列表，而 `source` 将是空字符串。

10.7. 全部放在一起

你已经了解很多基础的东西。让我们回来看看所有片段是如何整合到一起的。

作为开始，这里是一个接收命令行参数的脚本，它使用 `getopt` 模块。

```
 def main(argv):
...
     try:
         opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
     except getopt.GetoptError:
...
     for opt, arg in opts:
...
```

创建 `KantGenerator` 类的一个实例，然后将语法文件和源文件传给它，可能在命令行没有指定。

```
     k = KantGenerator(grammar, source)
```

`KantGenerator` 实例自动加载语法，它是一个 XML 文件。你使用自定义的 `openAnything` 函数打开这个文件 (可能保存在一个本地文件中或者一个远程服务器上)，然后使用内置的 `minidom` 解析函数将 XML 解析为一棵 Python 对象树。

```
     def _load(self, source):
         sock = toolbox.openAnything(source)
         xmldoc = minidom.parse(sock).documentElement
         sock.close()
```

哦，根据这种方式，你将使用到 XML 文档结构的知识建立一个引用的小缓冲，这些引用都只是 XML 文档中的元素。

```
     def loadGrammar(self, grammar):
         for ref in self.grammar.getElementsByTagName("ref"):
             self.refs[ref.attributes["id"].value] = ref
```

如果你在命令行中指定了某些源材料，你可以使用它；否则你将打开语法文件查找“顶层”引用 (没有被其它的东西引用) 并把它作为开始点。

```
     def getDefaultSource(self):
         xrefs = {}
         for xref in self.grammar.getElementsByTagName("xref"):
             xrefs[xref.attributes["id"].value] = 1
         xrefs = xrefs.keys()
         standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
         return '<xref id="%s"/>' % random.choice(standaloneXrefs)
```

现在你打开了了源材料。它是一个 XML，你每次解析一个节点。为了让代码分离并具备更高的可维护性，你可以使用针对每个节点类型的独立处理方法。

```
    def parse_Element(self, node):
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

你在语法里面跳来跳去，解析每一个 `p` 元素的所有孩子，

```
    def do_p(self, node):
...
        if doit:
            for child in node.childNodes: self.parse(child)
```

用任意一个孩子替换 `choice` 元素，

```
    def do_choice(self, node):
        self.parse(self.randomChildElement(node))
```

并用对应 `ref` 元素的任意孩子替换 `xref`，前面你已经进行了缓冲。

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

就这样一直解析，最后得到普通文本。

```
    def parse_Text(self, node):
        text = node.data
...
            self.pieces.append(text)
```

把结果打印出来。

```
 def main(argv):
...
    k = KantGenerator(grammar, source)
    print k.output()
```

10.8. 小结

Python 带有解析和操作 XML 文档非常强大的库。`minidom` 接收一个 XML 文件并将其解析为 Python 对象，并提供了对任意元素的随机访问。进一步，本章展示了如何利用 Python 创建一个“真实”独立的命令行脚本，连同命令行标志、命令行参数、错误处理，甚至从前一个程序的管道接收输入的能力。

在继续下一章前，你应该无困难地完成所有这些事情：

- 通过标准输入输出链接程序
- 使用 `getattr` 定义动态分发器
- 通过 `getopt` 使用命令行标志并进行验证

第 11 章 HTTP Web 服务

11.1. 概览

在讲解如何下载 web 页和如何从 URL 解析 XML时，你已经学习了关于 HTML 处理和 XML 处理，接下来让我们来更全面地探讨有关 HTTP web 服务的主题。

简单地讲，HTTP web 服务是指以编程的方式直接使用 HTTP 操作从远程服务器发送和接收数据。如果你要从服务器获取数据，直接使用 HTTP GET；如果您想发送新数据到服务器，使用 HTTP POST。(一些较高级的 HTTP web 服务 API 也定义了使用 HTTP PUT 和 HTTP DELETE 修改和删除现有数据的方法。) 换句话说，构建在 HTTP 协议中的 "verbs (动作)" (GET, POST, PUT 和 DELETE) 直接映射为接收、发送、修改和删除等应用级别的操作。

这种方法的主要优点是简单，并且许多不同的站点充分印证了这样的简单性是受欢迎的。数据 (通常是 XML 数据) 能静态创建和存储，或通过服务器端脚本和所有主流计算机语言 (包括用于下载数据的 HTTP 库) 动态生成。调试也很简单，因为您可以在任意浏览器中调用网络服务来查看这些原始数据。现代浏览器甚至可以为您进行良好的格式化并漂亮地打印这些 XML 数据，以便让您快速地浏览。

HTTP web 服务上的纯 XML 应用举例：

- Amazon API 允许您从 Amazon.com 在线商店获取产品信息。
- National Weather Service (美国) 和 Hong Kong Observatory (香港) 通过 web 服务提供天气警报。
- Atom API 用来管理基于 web 的内容。
- Syndicated feeds 应用于 weblogs 和新闻站点中带给您来自众多站点的最新消息。

在后面的几章里，我们将探索使用 HTTP 进行数据发送和接收传输的 API，但是不会将应用语义映射到潜在的 HTTP 语义。(所有这些都是通过 HTTP POST 这个管道完成的。) 但是本章将关注使用 HTTP GET 从远程服务器获取数据，并且将探索几个由纯 HTTP web 服务带来最大利益的 HTTP 特性。

如下所示为上一章曾经看到过的 `openanything` 模块的更高级版本：

例 11.1. `openanything.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
 import urllib2, urlparse, gzip
from StringIO import StringIO
USER_AGENT = 'OpenAnything/1.0 +http://diveintopython.org/http_web_services/'
class SmartRedirectHandler(urllib2.HTTPRedirectHandler):
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301(
            self, req, fp, code, msg, headers)
        result.status = code
        return result
```

```
    def http_error_302(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result
class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):
    def http_error_default(self, req, fp, code, msg, headers):
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code
        return result
def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    '''URL, filename, or string --> stream
    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner.  Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.
    If the etag argument is supplied, it will be used as the value of an
    If-None-Match request header.
    If the lastmodified argument is supplied, it must be a formatted
    date/time string in GMT (as returned in the Last-Modified header of
    a previous request).  The formatted date/time will be used
    as the value of an If-Modified-Since request header.
    If the agent argument is supplied, it will be used as the value of a
    User-Agent request header.
    '''
    if hasattr(source, 'read'):
        return source
    if source == '-':
        return sys.stdin
    if urlparse.urlparse(source)[0] == 'http':
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)
        if etag:
            request.add_header('If-None-Match', etag)
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)
        request.add_header('Accept-encoding', 'gzip')
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler())
        return opener.open(request)
    # try to open with native open function (if source is a filename)
    try:
        return open(source)
    except (IOError, OSError):
        pass
    # treat source as string
    return StringIO(str(source))
def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)
    result['data'] = f.read()
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')
        if f.headers.get('content-encoding', '') == 'gzip':
            # data came back gzip-compressed, decompress it
            result['data'] = gzip.GzipFile(fileobj=StringIO(result['data']])).read()
    if hasattr(f, 'url'):
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):
        result['status'] = f.status
    f.close()
    return result
```

进一步阅读

- Paul Prescod 认为纯 HTTP web 服务是 Internet 的未来。

11.2. 避免通过 HTTP 重复地获取数据

假如说你想用 HTTP 下载资源，例如一个 Atom feed 汇聚。你不仅仅想下载一次；而是想一次又一次地下载它，如每小时一次，从提供 news feed 的站点获得最新的消息。让我们首先用一种直接而原始的方法来实现它，然后看看如何改进它。

例 11.2. 用直接而原始的方法下载 feed

```
>>> import urllib
>>> data = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()
>>> print data
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
```

[1]	使用 Python 通过 HTTP 下载任何东西都简单得令人难以置信；实际上，只需要一行代码。 `urllib` 模块有一个便利的 `urlopen` 函数，它接受您所要获取的页面地址，然后返回一个类文件对象，您仅仅使用 `read()` 便可获得页面的全部内容。这再简单不过了。

那么这种方法有何不妥之处吗？当然，在测试或开发中一次性的使用没有什么不妥。我经常这样。我想要 feed 汇聚的内容，我就获取 feed 的内容。这种方法对其他 web 页面同样有效。但是一旦你开始按照 web 服务的方式去思考有规则的访问需求时 (记住，你说你计划每小时一次地重复获取这样的 feed) 就会发现这样的做法效率实在是太低了，并且对服务器来说也太笨了。

下面先谈论一些 HTTP 的基本特性。

11.3. HTTP 的特性

这里有五个你必须关注的 HTTP 重要特性。

11.3.1. 用户代理 (`User-Agent`)

`User-Agent` 是一种客户端告知服务器谁在什么时候通过 HTTP 请求了一个 web 页、feed 汇聚或其他类型的 web 服务的简单途径。当客户端请求一个资源时，应该尽可能明确发起请求的是谁，以便当产生异常错误时，允许服务器端的管理员与客户端的开发者取得联系。

默认情况下 Python 发送一个通用的 `User-Agent` ： `Python-urllib/1.15` 。下一节，您将看到更加有针对性的 `User-Agent` 。

11.3.2. 重定向 (Redirects)

有时资源移来移去。Web 站点重组内容，页面移动到了新的地址。甚至是 web 服务重组。原来位于 `http://example.com/index.xml` 的 feed 汇聚可能被移动到 `http://example.com/xml/atom.xml` 。或者因为一个机构的扩展或重组，整个域被迁移。例如， `http://www.example.com/index.xml` 可能被重定向到 `http://server-farm-1.example.com/index.xml` 。

您每次从 HTTP 服务器请求任何类型的资源时，服务器的响应中均包含一个状态代码。状态代码 `200` 的意思是 “一切正常，这就是您请求的页面”。状态代码 `404` 的意思是 “页面没找到”。 (当浏览 web 时，你可能看到过 404 errors。)

HTTP 有两种不同的方法表示资源已经被移动。状态代码 `302` 表示临时重定向；这意味着 “哎呀，访问内容被临时移动” (然后在 `Location:` 头信息中给出临时地址)。状态代码 `301` 表示永久重定向；这意味着 “哎呀，访问内容被永久移动” (然后在 `Location:` 头信息中给出新地址)。如果您获得了一个 `302` 状态代码和一个新地址，HTTP 规范说您应该使用新地址获取您的请求，但是下次您要访问同一资源时，应该使用原地址重试。但是如果您获得了一个 `301` 状态代码和一个新地址，您应该从此使用新地址。

当从 HTTP 服务器接受到一个适当的状态代码时，`urllib.urlopen` 将自动 “跟踪” 重定向，但不幸的是，当它做了重定向时不会告诉你。 你将最终获得所请求的数据，却丝毫不会察觉到在这个过程中一个潜在的库 “帮助” 你做了一次重定向操作。因此你将继续不断地使用旧地址，并且每次都将获得被重定向的新地址。这一过程要往返两次而不是一次：太没效率了！本章的后面，您将看到如何改进这一点，从而适当地且有效率地处理永久重定向。

11.3.3. `Last-Modified` / `If-Modified-Since`

有些数据随时都在变化。CNN.com 的主页经常几分钟就更新。另一方面，Google.com 的主页几个星期才更新一次 (当他们上传特殊的假日 logo，或为一个新服务作广告时)。 Web 服务是不变的：通常服务器知道你所请求的数据的最后修改时间，并且 HTTP 为服务器提供了一种将最近修改数据连同你请求的数据一同发送的方法。

如果你第二次 (或第三次，或第四次) 请求相同的数据，你可以告诉服务器你上一次获得的最后修改日期：在你的请求中发送一个 `If-Modified-Since` 头信息，它包含了上一次从服务器连同数据所获得的日期。如果数据从那时起没有改变，服务器将返回一个特殊的 HTTP 状态代码 `304`， 这意味着 “从上一次请求后这个数据没有改变”。这一点有何进步呢？当服务器发送状态编码 `304` 时，不再重新发送数据。您仅仅获得了这个状态代码。所以当数据没有更新时，你不需要一次又一次地下载相同的数据；服务器假定你有本地的缓存数据。

所有现代的浏览器都支持最近修改 (last-modified) 的数据检查。如果你曾经访问过某页，一天后重新访问相同的页时发现它没有变化，并奇怪第二次访问时页面加载得如此之快——这就是原因所在。你的浏览器首次访问时会在本地缓存页面内容，当你第二次访问，浏览器自动发送首次访问时从服务器获得的最近修改日期。服务器简单地返回 `304: Not Modified` (没有修改)，因此浏览器就会知道从本地缓存加载页面。在这一点上，Web 服务也如此智能。

Python 的 URL 库没有提供内置的最近修改数据检查支持，但是你可以为每一个请求添加任意的头信息并在每一个响应中读取任意头信息，从而自己添加这种支持。

11.3.4. `ETag` / `If-None-Match`

ETag 是实现与最近修改数据检查同样的功能的另一种方法：没有变化时不重新下载数据。其工作方式是：服务器发送你所请求的数据的同时，发送某种数据的 hash (在 `ETag` 头信息中给出)。hash 的确定完全取决于服务器。当第二次请求相同的数据时，你需要在 `If-None-Match:` 头信息中包含 ETag hash，如果数据没有改变，服务器将返回 `304` 状态代码。与最近修改数据检查相同，服务器仅仅 发送 `304` 状态代码；第二次将不为你发送相同的数据。在第二次请求时，通过包含 ETag hash，你告诉服务器：如果 hash 仍旧匹配就没有必要重新发送相同的数据，因为你还有上一次访问过的数据。

Python 的 URL 库没有对 ETag 的内置支持，但是在本章后面你将看到如何添加这种支持。

11.3.5. 压缩 (Compression)

最后一个重要的 HTTP 特性是 gzip 压缩。 关于 HTTP web 服务的主题几乎总是会涉及在网络线路上传输的 XML。XML 是文本，而且还是相当冗长的文本，而文本通常可以被很好地压缩。当你通过 HTTP 请求一个资源时，可以告诉服务器，如果它有任何新数据要发送给我时，请以压缩的格式发送。在你的请求中包含 `Accept-encoding: gzip` 头信息，如果服务器支持压缩，它将返回由 gzip 压缩的数据并且使用 `Content-encoding: gzip` 头信息标记。

Python 的 URL 库本身没有内置对 gzip 压缩的支持，但是你能为请求添加任意的头信息。Python 还提供了一个独立的 `gzip` 模块，它提供了对数据进行解压缩的功能。

注意我们用于下载 feed 汇聚的小单行脚本并不支持任何这些 HTTP 特性。让我们来看看如何改善它。

11.4. 调试 HTTP web 服务

首先，让我们开启 Python HTTP 库的调试特性并查看网络线路上的传输过程。这对本章的全部内容都很有用，因为你将添加越来越多的特性。

例 11.3. 调试 HTTP

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> import urllib
>>> feeddata = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/1.15
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 22:27:30 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

[1]	`urllib` 依赖于另一个 Python 的标准库， `httplib` 。通常你不必显式地给出 `import httplib` (`urllib` 会自动导入)，但是你可以为 `HTTPConnection` 类 (`urllib` 在内部使用它来访问 HTTP 服务器) 设置调试标记。这是一种令人难以置信的有用技术。Python 其他的一些库也有类似的调试标记，但是没有命名和开启它们的特殊标准；如果有类似的特性可用，你需要阅读每一个库的文档来查看使用方法。
[2]	既然已经设置了调试标记，HTTP 的请求和响应信息会实时地被打印出来。首先告诉你的是你连接服务器 `diveintomark.org` 的 80 端口，这是 HTTP 的标准端口。
[3]	当你请求 Atom feed 时， `urllib` 向服务器发送三行信息。第一行指出你使用的 HTTP verb 和资源的路径 (除去域名)。在本章中所有的请求都将使用 `GET` ，但是在下一章的 SOAP 中，你会看到所有的请求都使用 `POST` 。除了请求的动词不同之外，基本的语法是相同的。
[4]	第二行是 `Host` 头信息，它指出你所访问的服务的域名。这一点很重要，因为一个独立的 HTTP 服务器可以服务于多个不同的域。当前我的服务器服务于 12 个域；其他的服务器可以服务于成百乃至上千个域。
[5]	第三行是 `User-Agent` 头信息。在此你看到的是由 `urllib` 库默认添加的普通的 `User-Agent` 。在下一节，你会看到如何自定义它的更多细节。
[6]	服务器用状态代码和一系列的头信息答复 (其中一些数据可能会被存储到 `feeddata` 变量中)。这里的状态代码是 `200` ，意味着 “一切正常，这就是您请求的数据”。服务器也会告诉你响应请求的数据、一些有关服务器自身的信息，以及传给你的数据的内容类型。根据你的应用不同，这或许有用，或许没用。这充分确认了你所请求的是一个 Atom feed，瞧，你获得了 Atom feed (`application/atom+xml` ，它是已经注册的有关 Atom feeds 的内容类型)。
[7]	当此 Atom feed 有最近的修改，服务器会告诉你 (本例中，大约发生在 13 分钟之前)。当下次请求同样的 feed 时，你可以这个日期再发送给服务器，服务器将做最近修改数据检查。
[8]	服务器也会告诉你这个 Atom feed 有一个值为 `"e8284-68e0-4de30f80"` 的 ETag hash。这个 hash 自身没有任何意义；除了在下次访问相同的 feed 时将它送还给服务器之外，你也不需要用它做什么。然后服务器使用它告诉你修改日期是否被改变了。

11.5. 设置 `User-Agent`

改善你的 HTTP web 服务客户端的第一步就是用 `User-Agent` 适当地鉴别你自己。为了做到这一点，你需要远离基本的 `urllib` 而深入到 `urllib2` 。

例 11.4. `urllib2` 介绍

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener()
>>> feeddata = opener.open(request).read()
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:23:12 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

[1]	如果你的 Python IDE 仍旧为上一节的例子而打开着，你可以略过这一步，在开启 HTTP 调试时你能看到网络线路上的实际传输过程。
[2]	使用 `urllib2` 获取 HTTP 资源包括三个处理步骤，这会有助于你理解这一过程。第一步是创建 `Request` 对象，它接受一个你最终想要获取资源的 URL。注意这一步实际上还不能获取任何东西。
[3]	第二步是创建一个 URL 开启器 (opener)。它可以接受任何数量的处理器来控制响应的处理。但你也可以创建一个没有任何自定义处理器的开启器，在这儿你就是这么做的。你将在本章后面探究重定向的部分看到如何定义和使用自定义处理器的内容。
[4]	最后一个步骤是，使用你创建的 `Request` 对象告诉开启器打开 URL。因为你能从获得的信息中看到所有调试信息，这个步骤实际上获得了资源并且把返回数据存储在了 `feeddata` 中。

例 11.5. 给 `Request` 添加头信息

```
>>> request
<urllib2.Request instance at 0x00250AA8>
>>> request.get_full_url()
http://diveintomark.org/xml/atom.xml
>>> request.add_header('User-Agent',
`...` 'OpenAnything/1.0 +http://diveintopython.org/')
>>> feeddata = opener.open(request).read()
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: OpenAnything/1.0 +http://diveintopython.org/
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:45:17 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

[1]	继续前面的例子；你已经用你要访问的 URL 创建了 `Request` 。
[2]	使用 `Request` 对象的 `add_header` 方法，你能向请求中添加任意的 HTTP 头信息。第一个参数是头信息，第二个参数是头信息的值。 `User-Agent` 的约定格式是：应用名，跟一个斜线，跟版本号。剩下的是自由的格式，你将看到许多疯狂的变化，但通常这里应该包含你的应用的 URL。和你的请求的其他信息一样， `User-Agent` 会被服务器纪录下来，其中包含你的应用的 URL。如果发生错误，服务器管理员就能通过查看他们的访问日志与你联系。
[3]	之前你创建的 `opener` 对象也可以再生，且它将再次获得相同的 feed，但这次使用了你自定义的 `User-Agent` 头信息。
[4]	这就是你发送的自定义的 `User-Agent` ，代替了 Python 默认发送的一般的 `User-Agent` 。若你继续看，会注意到你定义的是 `User-Agent` 头信息，但实际上发送的是 `User-agent` 头信息。看看有何不同？ `urllib2` 改变了大小写所以只有首字母是大写的。这没问题，因为 HTTP 规定头信息的字段名是大小写无关的。

11.6. 处理 `Last-Modified` 和 `ETag`

既然你知道如何在你的 web 服务请求中添加自定义的 HTTP 头信息，接下来看看如何添加 `Last-Modified` 和 `ETag` 头信息的支持。

下面的这些例子将以调试标记置为关闭的状态来显示输出结果。如果你还停留在上一部分的开启状态，可以使用 `httplib.HTTPConnection.debuglevel = 0` 将其设置为关闭状态。或者，如果你认为有帮助也可以保持为开启状态。

例 11.6. 测试 `Last-Modified`

```
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener()
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.dict
{'date': 'Thu, 15 Apr 2004 20:42:41 GMT',
 'server': 'Apache/2.0.49 (Debian GNU/Linux)',
 'content-type': 'application/atom+xml',
 'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
 'etag': '"e842a-3e53-55d97640"',
 'content-length': '15955',
 'accept-ranges': 'bytes',
 'connection': 'close'}
>>> request.add_header('If-Modified-Since',
... firstdatastream.headers.get('Last-Modified'))
>>> seconddatastream = opener.open(request)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\urllib2.py", line 326, in open
    '_open', req)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 901, in http_open
    return self.do_open(httplib.HTTP, req)
  File "c:\python23\lib\urllib2.py", line 895, in do_open
    return self.parent.error('http', req, fp, code, msg, hdrs)
  File "c:\python23\lib\urllib2.py", line 352, in error
    return self._call_chain(*args)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 412, in http_error_default
    raise HTTPError(req.get_full_url(), code, msg, hdrs, fp)
urllib2.HTTPError: HTTP Error 304: Not Modified
```

[1]	还记得当调试标记设置为开启时所有那些你看到的 HTTP 头信息打印输出吗？这里便是用编程方式访问它们的方法： `firstdatastream.headers` 是一个类似 dictionary 行为的对象并且允许你获得任何个别的从 HTTP 服务器返回的头信息。
[2]	在第二次请求时，你用第一次请求获得的最近修改时间添加了 `If-Modified-Since` 头信息。如果数据没被改变，服务器应该返回一个 `304` 状态代码。
[3]	毫无疑问，数据没被改变。你可以从跟踪返回结果看到 `urllib2` 抛出了一个特殊异常， `HTTPError` ，以响应 `304` 状态代码。这有点不寻常，并且完全没有任何帮助。毕竟，它不是个错误；你明确地询问服务器如果没有变化就不要发送任何数据，并且数据没有变化，所以服务器告诉你它没有为你发送任何数据。那不是个错误；实际上也正是你所期望的。

`urllib2` 也为你认为是错误的其他条件引发 `HTTPError` 异常，比如 `404` (page not found)。实际上，它将为任何 除了状态代码 `200` (OK)、 `301` (permanent redirect)或 `302` (temporary redirect) 之外的状态引发 `HTTPError` 。捕获状态代码并简单返回它，而不是抛出异常，这应该对你很有帮助。为了实现它，你将需要自定义一个 URL 处理器。

例 11.7. 定义 URL 处理器

这个自定义的 URL 处理器是 `openanything.py` 的一部分。

```
class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):
    def http_error_default(self, req, fp, code, msg, headers):
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code
        return result
```

[1]	`urllib2` 是围绕 URL 处理器而设计的。每一个处理器就是一个能定义任意数量方法的类。当某事件发生时——比如一个 HTTP 错误，甚至是 `304` 代码——`urllib2` 审视用于处理它的一系列已定义的处理器方法。在此要用到自省，与 第 9 章 *XML* 处理中为不同节点类型定义不同处理器类似。但是 `urllib2` 是很灵活的，还可以内省为当前请求所定义的所有处理器。
[2]	当从服务器接收到一个 `304` 状态代码时， `urllib2` 查找定义的操作并调用 `http_error_default` 方法。通过定义一个自定义的错误处理，你可以阻止 `urllib2` 引发异常。取而代之的是，你创建 `HTTPError` 对象，返回它而不是引发异常。
[3]	这是关键部分：返回之前，你保存从 HTTP 服务器返回的状态代码。这将使你从主调程序轻而易举地访问它。

例 11.8. 使用自定义 URL 处理器

```
>>> request.headers
{'If-modified-since': 'Thu, 15 Apr 2004 19:45:21 GMT'}
>>> import openanything
>>> opener = urllib2.build_opener(
... openanything.DefaultErrorHandler())
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status
304
>>> seconddatastream.read()
''
```

[1]	继续前面的例子，`Request` 对象已经被设置，并且你已经添加了 `If-Modified-Since` 头信息。
[2]	这是关键所在：既然已经定义了你的自定义 URL 处理器，你需要告诉 `urllib2` 来使用它。还记得我怎么说 `urllib2` 将一个 HTTP 资源的访问过程分解为三个步骤的正当理由吗？这便是为什么构建 HTTP 开启器是其步骤之一，因为你能用你自定义的 URL 操作覆盖 `urllib2` 的默认行为来创建它。
[3]	现在你可以快速地打开一个资源，返回给你的对象既包括常规头信息 (使用 `seconddatastream.headers.dict` 访问它们)，也包括 HTTP 状态代码。在此，正如你所期望的，状态代码是 `304`，意味着此数据自从上次请求后没有被修改。
[4]	注意当服务器返回 `304` 状态代码时，并没有重新发送数据。这就是全部的关键：没有重新下载未修改的数据，从而节省了带宽。因此若你确实想要那个数据，你需要在首次获得它时在本地缓存数据。

处理 `ETag` 的工作也非常相似，只不过不是检查 `Last-Modified` 并发送 `If-Modified-Since`，而是检查 `ETag` 并发送 `If-None-Match`。让我们打开一个新的 IDE 会话。

例 11.9. 支持 `ETag` / `If-None-Match`

```
>>> import urllib2, openanything
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener(
... openanything.DefaultErrorHandler())
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.get('ETag')
'"e842a-3e53-55d97640"'
>>> firstdata = firstdatastream.read()
>>> print firstdata
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
>>> request.add_header('If-None-Match',
... firstdatastream.headers.get('ETag'))
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status
304
>>> seconddatastream.read()
''
```

[1]	使用 `firstdatastream.headers` 伪字典，你可以获得从服务器返回的 `ETag` (如果服务器没有返回 `ETag` 会发生什么？答案是，这一行代码将返回 `None` 。)
[2]	OK，你获得了数据。
[3]	现在进行第二次调用，将 `If-None-Match` 头信息设置为你第一次调用获得的 `ETag` 。
[4]	第二次调用静静地成功了 (没有出现任何的异常)，并且你又一次看到了从服务器返回的 `304` 状态代码。你第二次基于 `ETag` 发送请求，服务器知道数据没有被改变。
[5]	无论 `304` 是被 `Last-Modified` 数据检查还是 `ETag` hash 匹配触发的，获得 `304` 的同时都不会下载数据。这就是重点所在。

> 注意
> 在这些例子中，HTTP 服务器同时支持 `Last-Modified` 和 `ETag` 头信息，但并非所有的服务器皆如此。作为一个 web 服务的客户端，你应该为支持两种头信息做准备，但是你的程序也应该为服务器仅支持其中一种头信息或两种头信息都不支持而做准备。

11.7. 处理重定向

你可以使用两种不同的自定义 URL 处理器来处理永久重定向和临时重定向。

首先，让我们来看看重定向处理的必要性。

例 11.10. 没有重定向处理的情况下，访问 web 服务

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> request = urllib2.Request(
... 'http://diveintomark.org/redir/example301.xml')
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.url
'http://diveintomark.org/xml/atom.xml'
>>> f.headers.dict
{'content-length': '15955',
'accept-ranges': 'bytes',
'server': 'Apache/2.0.49 (Debian GNU/Linux)',
'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
'connection': 'close',
'etag': '"e842a-3e53-55d97640"',
'date': 'Thu, 15 Apr 2004 22:06:25 GMT',
'content-type': 'application/atom+xml'}
>>> f.status
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: addinfourl instance has no attribute 'status'
```

[1]	你最好开启调试状态，看看发生了什么。
[2]	这是一个我已经设置了永久重定向到我的 Atom feed `http://diveintomark.org/xml/atom.xml` 的 URL。
[3]	毫无疑问，当你试图从那个地址下载数据时，服务器会返回 `301` 状态代码，告诉你你访问的资源已经被永久移动了。
[4]	服务器同时返回 `Location:` 头信息，它给出了这个数据的新地址。
[5]	`urllib2` 注意到了重定向状态代码并会自动从 `Location:` 头信息中给出的新地址获取数据。
[6]	从 `opener` 返回的对象包括新的永久地址和第二次请求获得的所有头信息 (从一个新的永久地址获得)。但是状态代码不见了，因此你无从知晓重定向到底是永久重定向还是临时重定向。这是至关重要的：如果这是临时重定向，那么你应该继续使用旧地址访问数据。但是如果是永久重定向 (正如本例)，你应该从现在起使用新地址访问数据。

这不太理想，但很容易改进。实际上当 `urllib2` 遇到 `301` 或 `302` 时的行为并不是我们所期望的，所以让我们来覆盖这些行为。如何实现呢？用一个自定义的处理器，正如你处理 `304` 代码所做的。

例 11.11. 定义重定向处理器

这个类定义在 `openanything.py` 。

```
class SmartRedirectHandler(urllib2.HTTPRedirectHandler):
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301(
            self, req, fp, code, msg, headers)
        result.status = code
        return result
    def http_error_302(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result
```

[1]	重定向行为定义在 `urllib2` 的一个叫做 `HTTPRedirectHandler` 的类中。我们不想完全地覆盖这些行为，只想做点扩展，所以我们子类化 `HTTPRedirectHandler` ，从而我们仍然可以调用祖先类来实现所有原来的功能。
[2]	当从服务器获得 `301` 状态代码， `urllib2` 将搜索处理器并调用 `http_error_301` 方法。我们首先要做的就是在祖先中调用 `http_error_301` 方法，它将处理查找 `Location:` 头信息的工作并跟踪重定向到新地址。
[3]	这是关键：返回之前，你存储了状态代码 (`301`)，所以主调程序稍后就可以访问它了。
[4]	临时重定向 (状态代码 `302`) 以相同的方式工作：覆盖 `http_error_302` 方法，调用祖先，并在返回之前保存状态代码。

这将为我们带来什么？现在你可以用自定义重定向处理器构造一个的 URL 开启器，并且它依然能自动跟踪重定向，也能展示出重定向状态代码。

例 11.12. 使用重定向处理器检查永久重定向

```
>>> request = urllib2.Request('http://diveintomark.org/redir/example301.xml')
>>> import openanything, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> opener = urllib2.build_opener(
... openanything.SmartRedirectHandler())
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: 'GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.status
301
>>> f.url
'http://diveintomark.org/xml/atom.xml'
```

[1]	首先，用刚刚定义的重定向处理器创建一个 URL 开启器。
[2]	你发送了一个请求，并在响应中获得了 `301` 状态代码。 如此一来， `http_error_301` 方法就被调用了。你调用了祖先类，跟踪了重定向并且发送了一个新地址（`http://diveintomark.org/xml/atom.xml`）请求。
[3]	这是决定性的一步：现在，你不仅做到了访问一个新 URL，而且获得了重定向的状态代码，所以你可以断定这是一个永久重定向。下一次你请求这个数据时，就应该使用 `f.url` 指定的新地址（`http://diveintomark.org/xml/atom.xml`）。如果你已经在配置文件或数据库中存储了这个地址，就需要更新旧地址而不是反复地使用旧地址请求服务。现在是更新你的地址簿的时候了。

同样的重定向处理也可以告诉你不该 更新你的地址簿。

例 11.13. 使用重定向处理器检查临时重定向

```
>>> request = urllib2.Request(
... 'http://diveintomark.org/redir/example302.xml')
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example302.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 302 Found\r\n'
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 314
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.status
302
>>> f.url
http://diveintomark.org/xml/atom.xml
```

[1]	这是一个 URL，我已经设置了它，让它告诉客户端临时 重定向到 `http://diveintomark.org/xml/atom.xml` 。
[2]	服务器返回 `302` 状态代码，标识出一个临时重定向。数据的临时新地址在 `Location:` 头信息中给出。
[3]	`urllib2` 调用你的 `http_error_302` 方法，它调用了 `urllib2.HTTPRedirectHandler` 中的同名的祖先方法，跟踪重定向到一个新地址。然后你的 `http_error_302` 方法存储状态代码 (`302`) 使主调程序在稍后可以获得它。
[4]	此时，已经成功追踪重定向到 `http://diveintomark.org/xml/atom.xml` 。 `f.status` 告诉你这是一个临时重定向，这意味着你应该继续使用原来的地址 (`http://diveintomark.org/redir/example302.xml`) 请求数据。也许下一次它仍然被重定向，也许不会。也许会重定向到不同的地址。这也不好说。服务器说这个重定向仅仅是临时的，你应该尊重它。并且现在你获得了能使主调程序尊重它的充分信息。

11.8. 处理压缩数据

你要支持的最后一个重要的 HTTP 特性是压缩。许多 web 服务具有发送压缩数据的能力，这可以将网络线路上传输的大量数据消减 60% 以上。这尤其适用于 XML web 服务，因为 XML 数据 的压缩率可以很高。

服务器不会为你发送压缩数据，除非你告诉服务器你可以处理压缩数据。

例 11.14. 告诉服务器你想获得压缩数据

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> request.add_header('Accept-encoding', 'gzip')
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
Accept-encoding: gzip
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:24:39 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Vary: Accept-Encoding
header: Content-Encoding: gzip
header: Content-Length: 6289
header: Connection: close
header: Content-Type: application/atom+xml
```

[1]	这是关键：一创建了 `Request` 对象，就添加一个 `Accept-encoding` 头信息告诉服务器你能接受 gzip 压缩数据。`gzip` 是你使用的压缩算法的名称。理论上你可以使用其它的压缩算法，但是 `gzip` 是 web 服务器上使用率高达 99% 的一种。
[2]	这是你的头信息传越网络线路的过程。
[3]	这是服务器的返回信息：`Content-Encoding: gzip` 头信息意味着你要回得的数据已经被 gzip 压缩了。
[4]	`Content-Length` 头信息是已压缩数据的长度，并非解压缩数据的长度。一会儿你会看到，实际的解压缩数据长度为 15955，因此 gzip 压缩节省了 60% 以上的网络带宽！

例 11.15. 解压缩数据

```
>>> compresseddata = f.read()
>>> len(compresseddata)
6289
>>> import StringIO
>>> compressedstream = StringIO.StringIO(compresseddata)
>>> import gzip
>>> gzipper = gzip.GzipFile(fileobj=compressedstream)
>>> data = gzipper.read()
>>> print data
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
>>> len(data)
15955
```

[1]	继续上面的例子，`f` 是一个从 URL 开启器返回的类文件对象。使用它的 `read()` 方法将正常地获得非压缩数据，但是因为这个数据已经被 gzip 压缩过，所以这只是获得你想要的最终数据的第一步。
[2]	好吧，只是先得有点儿凌乱的步骤。Python 有一个 `gzip` 模块，它能读取 (当然也能写入) 磁盘上的 gzip 压缩文件。但是磁盘上还没有文件，只在内存里有一个 gzip 压缩缓冲区，并且你不想仅仅为了解压缩而写出一个临时文件。那么怎么做来从内存数据 (`compresseddata`) 创建类文件对象呢？这需要使用 `StringIO` 模块。你首次看到 `StringIO` 模块是在上一章，但现在你会发现它的另一种用法。
[3]	现在你可以创建 `GzipFile` 的一个实例，并且告诉它其中的 “文件” 是一个类文件对象 `compressedstream` 。
[4]	这是做所有工作的一行：从 `GzipFile` 中 “读取” 将会解压缩数据。感到奇妙吗？是的，它确实解压缩了数据。`gzipper` 是一个类文件对象，它代表一个 gzip 压缩文件。尽管这个 “文件” 并非一个磁盘上的真实文件；但 `gzipper` 还是从你用 `StringIO` 包装了压缩数据的类文件对象中 “读取” 数据，而它仅仅是内存中的变量 `compresseddata` 。压缩的数据来自哪呢？最初你从远程 HTTP 服务器下载它，通过从用 `urllib2.build_opener` 创建的类文件对象中 “读取”。令人吃惊吧，这就是所有的步骤。链条上的每一步都完全不知道上一步在造假。
[5]	看看吧，实际的数据 (实际为 15955 bytes)。

“等等!” 我听见你在叫。“还能更简单吗！” 我知道你在想什么。你在，既然 `opener.open` 返回一个类文件对象，那么为什么不抛弃中间件 `StringIO` 而通过 `f` 直接访问 `GzipFile` 呢？OK，或许你没想到，但是别为此担心，因为那样无法工作。

例 11.16. 从服务器直接解压缩数据

```
>>> f = opener.open(request)
>>> f.headers.get('Content-Encoding')
'gzip'
>>> data = gzip.GzipFile(fileobj=f).read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\gzip.py", line 217, in read
    self._read(readsize)
  File "c:\python23\lib\gzip.py", line 252, in _read
    pos = self.fileobj.tell()   # Save current position
AttributeError: addinfourl instance has no attribute 'tell'
```

[1]	继续前面的例子，你已经有一个设置了 `Accept-encoding: gzip` 头信息的 `Request` 对象。
[2]	简单地打开请求将获得你的头信息 (虽然还没下载任何数据)。正如你从 `Content-Encoding` 头信息所看到的，这个数据被要求用 gzip 压缩发送。
[3]	从 `opener.open` 返回了一个类文件对象，从头信息中你可以获知，你将获得 gzip 压缩数据。为什么不简单地通过那个类文件对象直接访问 `GzipFile` 呢？当你从 `GzipFile` 实例 “读取” 时，它将从远程 HTTP 服务器 “读取” 被压缩的数据并且立即解压缩。这是个好主意，但是不行。由 gzip 压缩的工作方式所致，`GzipFile` 需要存储其位置并在压缩文件上往返游走。当 “文件” 是来自远程服务器的字节流时无法工作；你能用它做的所有工作就是一次返回一个字节流，而不是在字节流上往返。所以使用 `StringIO` 这种看上去不太优雅的手段是最好的解决方案：下载压缩的数据，用 `StringIO` 创建一个类文件对象，并从中解压缩数据。

11.9. 全部放在一起

你已经看到了构造一个智能的 HTTP web 客户端的所有片断。现在让我们看看如何将它们整合到一起。

例 11.17. `openanything` 函数

这个函数定义在 `openanything.py` 中。

```
def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    # non-HTTP code omitted for brevity
    if urlparse.urlparse(source)[0] == 'http':
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)
        if etag:
            request.add_header('If-None-Match', etag)
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)
        request.add_header('Accept-encoding', 'gzip')
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler())
        return opener.open(request)
```

[1]	`urlparse` 是一个解析 URL 的便捷的工具模块。它的主要函数也叫 `urlparse`，接受一个 URL 并将其拆分为 tuple (scheme (协议), domain (域名), path (路径), params (参数), query string parameters (请求字符串参数), fragment identifier (片段效验符))。当然，你唯一需要注意的就是 scheme，确认你处理的是一个 HTTP URL (`urllib2` 才能处理)。
[2]	通过调用函数使用 `User-Agent` 向 HTTP 服务器确定你的身份。如果没有 `User-Agent` 被指定，你会使用一个默认的，就是定义在早期的 `openanything.py` 模块中的那个。你从来不会使用到默认的定义在 `urllib2` 中的那个。
[3]	如果给出了 `ETag`，要在 `If-None-Match` 头信息中发送它。
[4]	如果给出了最近修改日期，要在 `If-Modified-Since` 头信息中发送它。
[5]	如果可能要告诉服务器你要获取压缩数据。
[6]	使用两个 自定义 URL 处理器创建一个 URL 开启器： `SmartRedirectHandler` 终于处理 `301` 和 `302` 重定向，而 `DefaultErrorHandler` 用于处理 `304`，`404` 以及其它的错误条件。
[7]	就是这样！打开 URL 并返回一个类文件对象给调用者。

例 11.18. `fetch` 函数

这个函数定义在 `openanything.py` 中。

```
def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)
    result['data'] = f.read()
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')
        if f.headers.get('content-encoding', '') == 'gzip':
            # data came back gzip-compressed, decompress it
            result['data'] = gzip.GzipFile(fileobj=StringIO(result['data'])).read()
    if hasattr(f, 'url'):
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):
        result['status'] = f.status
    f.close()
    return result
```

[1]	首先，你用 URL、 `ETag` hash、 `Last-Modified` 日期和 `User-Agent` 调用 `openAnything` 函数。
[2]	读取从服务器返回的真实数据。这可能是被压缩的；如果是，将在后面进行解压缩。
[3]	保存从服务器返回的 `ETag` hash，这样主调程序下一次就能把它传递给你，然后再传递给 `openAnything` ，放到 `If-None-Match` 头信息里发送给远程服务器。
[4]	也要保存 `Last-Modified` 数据。
[5]	如果服务器说它发送的是压缩数据，就执行解压缩。
[6]	如果你的服务器返回一个 URL 就保存它，并在查明之前假定状态代码为 `200` 。
[7]	如果其中一个自定义 URL 处理器捕获了一个状态代码，也要保存下来。

例 11.19. 使用 `openanything.py`

```
>>> import openanything
>>> useragent = 'MyHTTPWebServicesApp/1.0'
>>> url = 'http://diveintopython.org/redir/example301.xml'
>>> params = openanything.fetch(url, agent=useragent)
>>> params
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': 'Thu, 15 Apr 2004 19:45:21 GMT',
'etag': '"e842a-3e53-55d97640"',
'status': 301,
'data': '<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
<-- rest of data omitted for brevity -->'}
>>> if params['status'] == 301:
... url = params['url']
>>> newparams = openanything.fetch(
... url, params['etag'], params['lastmodified'], useragent)
>>> newparams
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': None,
'etag': '"e842a-3e53-55d97640"',
'status': 304,
'data': ''}
```

[1]	第一次获取资源时，你没有 `ETag` hash 或 `Last-Modified` 日期，所以你不用使用这些参数。 (它们是可选参数。)
[2]	你获得了一个 dictionary，它包括几个有用的头信息、HTTP 状态代码和从服务器返回的真实数据。 `openanything` 在内部处理 gzip 压缩；在本级别上你不必关心它。
[3]	如果你得到一个 `301` 状态代码，表示是个永久重定向，你需要把你的 URL 更新为新地址。
[4]	第二次获取相同的资源时，你已经从以往获得了各种信息：URL (可能被更新了)、从上一次访问获得的 `ETag` 、从上一次访问获得的 `Last-Modified` 日期，当然还有 `User-Agent` 。
[5]	你重新获取了这个 dictionary，但是数据没有改变，所以你得到了一个 `304` 状态代码而没有数据。

11.10. 小结

`openanything.py` 及其函数现在可以完美地工作了。

每个客户端都应该支持 HTTP web 服务的以下 5 个重要特性：

- 通过设置适当的 `User-Agent` 识别你的应用。
- 适当地处理永久重定向。
- 支持 `Last-Modified` 日期检查从而避免在数据未改变的情况下重新下载数据。
- 支持 `ETag` hash 从而避免在数据未改变的情况下重新下载数据。
- 支持 gzip 压缩从而在数据已经 改变的情况下尽可能地减少传输带宽。

第 12 章 SOAP Web 服务

第 11 章 关注 HTTP 上面向文档的web 服务。“输入参数” 是 URL，“返回值” 是需要你来解析的一个实际的 XML 文档。

本章将关注更加结构化的 SOAP web 服务。SOAP 不需要你直接与 HTTP 请求和 XML 文档打交道，而是允许你模拟返回原始数据类型的函数调用。正像你将要看到的，这个描述恰如其份；你可以使用标准 Python 调用语法通过 SOAP 库去调用一个函数，这个函数也自然会返回 Python 对象和值。但揭开这层面纱，SOAP 库实际上执行了一个多个 XML 文档和远程服务器参与的复杂处理过程。

SOAP 的贴切定义很复杂，不要误认为 SOAP 就是用于调用远程函数。有些人觉得应该补充上：SOAP 还允许单向异步的信息通过，以及面向文档的 Web 服务。有这样想法的人是正确的，SOAP 的确是这样，但却不止于此。但这一章的重点在于所谓的 “RPC-style” SOAP——调用远程函数获得返回结果。

12.1. 概览

你用 Google，对吧？它是一个很流行的搜索引擎。你是否希望能以程序化的方式访问 Google 的搜索结果呢？现在你能做到了。下面是一个用 Python 搜索 Google 的程序。

例 12.1. `search.py`

```
from SOAPpy import WSDL
# you'll need to configure these two values;
# see http://www.google.com/apis/
WSDLFILE = '/path/to/copy/of/GoogleSearch.wsdl'
APIKEY = 'YOUR_GOOGLE_API_KEY'
_server = WSDL.Proxy(WSDLFILE)
def search(q):
    """Search Google and return list of {title, link, description}"""
    results = _server.doGoogleSearch(
        APIKEY, q, 0, 10, False, "", False, "", "utf-8", "utf-8")
    return [{"title": r.title.encode("utf-8"),
             "link": r.URL.encode("utf-8"),
             "description": r.snippet.encode("utf-8")}
            for r in results.resultElements]
if __name__ == '__main__':
    import sys
    for r in search(sys.argv[1])[:5]:
        print r['title']
        print r['link']
        print r['description']
        print
```

你可以在较大的程序中以模块导入并使用它，也可以在命令行上运行这个脚本。在命令行上，需要把查询字符串作为命令行参数使用，之后就会打印出最前面的五个 Google 查询结果，包括：URL、标题和描述信息。

下面是以 “python” 作为命令行参数的查询结果。

例 12.2. `search.py` 的使用样例

```
C:\diveintopython\common\py> python search.py "python"
<b>Python</b> Programming Language
http://www.python.org/
Home page for <b>Python</b>, an interpreted, interactive, object-oriented,
extensible<br> programming language. <b>...</b> <b>Python</b>
is OSI Certified Open Source: OSI Certified.
<b>Python</b> Documentation Index
http://www.python.org/doc/
 <b>...</b> New-style classes (aka descrintro). Regular expressions. Database
API. Email Us.<br> docs@<b>python</b>.org. (c) 2004\. <b>Python</b>
Software Foundation. <b>Python</b> Documentation. <b>...</b>
Download <b>Python</b> Software
http://www.python.org/download/
Download Standard <b>Python</b> Software. <b>Python</b> 2.3.3 is the
current production<br> version of <b>Python</b>. <b>...</b>
<b>Python</b> is OSI Certified Open Source:
Pythonline
http://www.pythonline.com/
Dive Into <b>Python</b>
http://diveintopython.org/
Dive Into <b>Python</b>. <b>Python</b> from novice to pro. Find:
<b>...</b> It is also available in multiple<br> languages. Read
Dive Into <b>Python</b>. This book is still being written. <b>...</b>
```

进一步阅读

- http://www.xmethods.net/ 是一个访问 SOAP web 服务的公共知识库。
- SOAP 规范相当可读，如果你喜欢这类东西的话。

12.2. 安装 SOAP 库

与本书中的其他代码不同，本章依赖的库不是 Python 预安装的。

在深入学习 SOAP web 服务之前，你需要安装三个库：PyXML、fpconst 和 SOAPpy。

12.2.1. 安装 PyXML

你要用到的第一个库是 PyXML，它是 XML 库的一个高级组件，提供了比我们在 第 9 章 学习的 XML 内建库更多的功能。

过程 12.1.

下面是安装 PyXML 的步骤：

1. 访问 http://pyxml.sourceforge.net/，点击 Downloads，下载适合你所使用操作系统的最新版本。
2. 如果你所使用的是 Windows，那么你有多个选择。一定要确保你所下载的 PyXML 和你所使用的 Python 版本匹配。
3. 双击安装程序。如果你下载的是为 Windows 提供的 PyXML 0.8.3，并且你所使用的是 Python 2.3，这个安装程序应该是 `PyXML-0.8.3.win32-py2.3.exe`。
4. 深入安装过程。
5. 安装完成后，关闭安装程序，没有任何安装成功的昭示 (并没有在开始菜单、快捷栏或桌面出现图标)。因为 PyXML 仅仅是被其他程序调用的 XML 的库集合。

要检验 PyXML 安装得是否正确，可以运行 Python IDE，下面的指令可以看到 XML 库的安装版本。

例 12.3. 检验 PyXML 安装

```
>>> import xml
>>> xml.__version__
'0.8.3'
```

这个安装版本号应该和你所下载并安装的 PyXML 安装程序版本号一致。

12.2.2. 安装 fpconst

你所需要安装的第二个库是 fpconst，它是一系列支持 IEEE754 double-precision 特殊值的常量和函数，提供了对 Not-a-Number (NaN), Positive Infinity (Inf) 和 Negative Infinity (-Inf) 等特殊值的支持，而这是 SOAP 数据类型规范的组成部分。

过程 12.2.

下面是 fpconst 的安装过程：

1. 从 http://www.analytics.washington.edu/statcomp/projects/rzope/fpconst/ 下载 fpconst 的最新版本。
2. 提供了两种格式的下载：`.tar.gz` 和 `.zip`。如果你使用的是 Windows 操作系统，下载 `.zip` 文件；其他情况下应该下载 `.tar.gz` 文件。
3. 对这个文件进行解压缩。在 Windows XP 上你可以鼠标右键单击这个文件并选择“解压文件”；在较早的 Windows 版本上则需要 WinZip 之类的第三方解压程序。在 Mac OS X 上，可以右键单击压缩文件进行解压。
4. 打开命令提示符窗口并定位到解压目录。
5. 键入 **`python setup.py install`** 运行安装程序。

要检验 fpconst 安装得是否正确，运行 Python IDE 并查看版本号。

例 12.4. 检验 fpconst 安装

```
>>> import fpconst
>>> fpconst.__version__
'0.6.0'
```

这个安装版本号应该和你所下载并用于安装的 fpconst 压缩包版本号一致。

12.2.3. 安装 SOAPpy

第三个，也是最后一个需要安装的库是 SOAP 库本身：SOAPpy。

过程 12.3.

下面是安装 SOAPpy的过程：

1. 访问 http://pywebsvcs.sourceforge.net/ 并选择 SOAPpy 部分中最新的官方发布。
2. 提供了两种格式的下载。如果你使用的是 Windows，那么下载 `.zip` 文件；其他情况则下载 `.tar.gz` 文件。
3. 和安装 fpconst 时一样先解压下载的文件.
4. 打开命令提示符窗口并定位到解压 SOAPpy 文件的目录。
5. 键入 **`python setup.py install`** 运行安装程序。

要检验 SOAPpy 安装得是否正确，运行 Python IDE 并查看版本号。

例 12.5. 检验 SOAPpy 安装

```
>>> import SOAPpy
>>> SOAPpy.__version__
'0.11.4'
```

这个安装版本号应该和你所下载并用于安装的 SOAPpy 压缩包版本号一致。

12.3. 步入 SOAP

调用远程函数是 SOAP 的核心功能。有很多提供公开 SOAP 访问的服务器提供用于展示的简单功能。

最受欢迎的 SOAP 公开访问服务器是 http://www.xmethods.net/。这个例子使用了一个展示函数，可以根据美国邮政编码返回当地气温。

例 12.6. 获得现在的气温

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> namespace = 'urn:xmethods-Temperature'
>>> server = SOAPProxy(url, namespace)
>>> server.getTemp('27502')
80.0
```

[1]	你通过 `SOAPProxy` 这个代理 (proxy) 类访问远程 SOAP 服务器。这个代理处理了所有的 SOAP 内部事务，其中包括：根据函数名和参数列表创建 XML 请求文档，并将这个请求文档通过 HTTP 发送到远程 SOAP 服务器；解析 XML 返回文档，并创建本地的 Python 返回值。在下一节中你将看到这个 XML 文档。
[2]	每个 SOAP 服务都有一个 URL 用以处理所有请求。相同的 URL 可以用于所有的函数请求。每个特定服务则只有一个函数。但稍后你将看到的 Google API 却有多个函数。这个服务的 URL 提供给所有函数分享。每个 SOAP 服务都有一个命名空间 (namespace)，这个命名空间是由服务器任意命名的。这不过是为调用 SOAP 方法设置的。它使得服务器让多个不相关的服务共享服务 URL 和路径请求成为可能。这与 Python 中模块相对于包的关系类似。
[3]	这里你创建了包含服务 URL 和服务命名空间的 `SOAPProxy`。此时还不会连接到 SOAP 服务器；仅仅是建立了一个本地 Python 对象。
[4]	到此为止，如果你的设置完全正确，应该可以向调用本地函数一样调用远程 SOAP 方法。这和给普通函数传递参数并接收返回值一样，但在背后却隐藏着很多的工作。

让我们看一看这些背后的工作。

12.4. SOAP 网络服务查错

SOAP 提供了一个很方便的方法用以查看背后的情形。

`SOAPProxy` 的两个小设置就可以打开查错模式。

例 12.7. SOAP 网络服务查错

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> n = 'urn:xmethods-Temperature'
>>> server = SOAPProxy(url, namespace=n)
>>> server.config.dumpSOAPOut = 1
>>> server.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('27502')
*** Outgoing SOAP ******************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
*** Incoming SOAP ******************************************************
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">80.0</return>
</ns1:getTempResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
>>> temperature
80.0
```

[1]	首先，和平常一样，建立带有服务 URL 和命名空间的 `SOAPProxy`。
[2]	然后，通过设置 `server.config.dumpSOAPIn` 和 `server.config.dumpSOAPOut` 打开查错模式。
[3]	最后，和平常一样，调用远程 SOAP 方法。SOAP 库将会输出送出的 XML 请求文档和收到的 XML 返回文档。这是 `SOAPProxy` 为你做的所有工作。有点恐怖，不是吗？让我们来分析一下。

大部分 XML 请求文档都基于模板文件。忽略所有命名空间声明这些对于所有 SOAP 调用都一成不变的东西。这个 “函数调用” 的核心是 <Body> 当中的部分：

```
<ns1:getTemp
  xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>
</ns1:getTemp>
```

[1]	这个元素名 `getTemp` 就是函数名。`SOAPProxy` 使用 `getattr` 作为分发器。有别于使用方法名分别调用本地方法，这里使用方法名构造了一个 XML 请求文档。
[2]	函数的 XML 元素被存储于一个特别的命名空间，这个命名空间就是你在建立 `SOAPProxy` 对象时所指定的那个命名空间。也不必为 `SOAP-ENC:root` 而苦恼，因为它也是基于模板文件的。
[3]	函数的参数也被记入 XML 文档。`SOAPProxy` 查看并确定每个参数的数据类型 (这里是 string 字符串类型)。参数的数据类型记入 `xsi:type` 属性，并在其后记入实际的字符串值。

返回的 XML 文档同样容易理解，重点在于知道应该忽略掉哪些内容。把注意力集中在 <Body> 部分：

```
<ns1:getTempResponse
  xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">80.0</return>
</ns1:getTempResponse>
```

[1]	服务器传回的值记录在 <getTempResponse> 部分的几行中。通常包括函数名和回应 `(Response)`。当然其他的内容也可能出现在这里，但 `SOAPProxy` 所重视的不是这里的元素名，而是命名空间。
[2]	服务器返回时所使用的命名空间就是在请求时所用的命名空间，也就是在创建 `SOAPProxy` 对象时所指定的命名空间。本章稍后的部分中，我们将看到在创建 `SOAPProxy` 对象时忘记指定功能名空间会怎样。
[3]	这是返回值和它的数据类型 (浮点类型 float)。`SOAPProxy` 使用显式数据类型创建一个本地数据类型的 Python 对象并返回之。

12.5. WSDL 介绍

`SOAPProxy` 类本地方法调用并透明地转向到远程 SOAP 方法。正如你所看到的，这是很多的工作，`SOAPProxy` 快速和透明地完成他们。它没有做到的是提供方法自省的手段。

考虑一下：前面两部分所展现的调用只有一个参数和返回的简单远程 SOAP 方法。服务 URL 和一系列参数及它们的数据类型需要被知道并跟踪。任何的缺失或错误都会导致整体的失败。

这并没有什么可惊讶的。如果我要调用一个本地函数，我需要知道函数所在的包和模块名 (与之对应的则是服务 URL 和命名空间)。我还需要知道正确的函数名以及其函数个数。Python 精妙地不需明示类型，但我还是需要知道有多少个参数需要传递，多少个值将被返回。

最大的区别就在于内省。就像你在 第 4 章 看到的那样，Python 擅长于让你实时地去探索模块和函数的情况。你可以对一个模块中的所有函数进行列表，并不费吹灰之力地明了函数的声明和参数情况。

WSDL 允许你对 SOAP 网络服务做相同的事情。WSDL 是 “网络服务描述语言 (Web Services Description Language)”的缩写。它尽管是为自如地表述多种类型的网络服务而设定，却也经常用于描述 SOAP 网络服务。

一个 WSDL 文件不过就是一个文件。更具体地讲，是一个 XML 文件。通常存储于你所访问的 SOAP 网络服务这个被描述对象所在的服务器上，并没有什么特殊之处。在本章稍后的位置，我们将下载 Google API 的 WSDL 文件并在本地使用它。这并不意味着本地调用 Google，这个 WSDL 文件所描述的仍旧是 Google 服务器上的远程函数。

在 WSDL 文件中描述了调用相应的 SOAP 网络服务的一切：

- 服务 URL 和命名空间
- 网络服务的类型 (可能是 SOAP 的函数调用，但我说过，WSDL 足够自如地去描述网络服务的广泛内容)
- 有效函数列表
- 每个函数的参数
- 每个参数的类型
- 每个函数的返回值及其数据类型

换言之，一个 WSDL 文件告诉你调用 SOAP 所需要知道的一切。

12.6. 以 WSDL 进行 SOAP 内省

就像网络服务舞台上的所有事物，WSDL 也经历了一个充满明争暗斗而且漫长多变的历史。我不打算讲述这段令我伤心的历史。还有一些其他的标准提供相同的支持，但 WSDL 还是胜出，所以我们还是来学习一下如何使用它。

WSDL 最基本的功能便是让你揭示 SOAP 服务器所提供的有效方法。

例 12.8. 揭示有效方法

```
>>> from SOAPpy import WSDL
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> server.methods.keys()
[u'getTemp']
```

[1]	SOAPpy 包含一个 WSDL 解析器。在本书写作之时，它被标示为开发的初级阶段，但我从来没有在解析任何 WSDL 文件时遇到问题。
[2]	使用一个 WSDL 文件，你还是要用到一个 proxy 类： `WSDL.Proxy` ，它只需一个参数：WSDL 文件。我指定的是存储在远程服务器上的 WSDL 的 URL，但是这个 proxy 类对于本地的 WSDL 副本工作同样出色。创建 WSDL proxy 将会下载 WSDL 文件并解析它，所以如果 WSDL 文件有任何问题 (或者由于网络问题不能获得) 你会立刻知道。
[3]	WSDL proxy 类通过 Python 字典 `server.methods` 揭示有效函数。所以列出有效方法只需调用字典方法 `keys()` 。

好的，你知道这个 SOAP 服务器提供一个方法： `getTemp` 。但是如何去调用它呢？WSDL 也在这方面提供信息。

例 12.9. 揭示一个方法的参数

```
>>> callInfo = server.methods['getTemp']
>>> callInfo.inparams
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AD0>]
>>> callInfo.inparams[0].name
u'zipcode'
>>> callInfo.inparams[0].type
(u'http://www.w3.org/2001/XMLSchema', u'string')
```

[1]	`server.methods` 字典中记录一个 SOAPpy 的特别结构，被称为 `CallInfo` 。 `CallInfo` 对象中包含着特定函数和函数参数的信息。
[2]	函数参数信息存储在 `callInfo.inparams` 中，这是一个记录每一个参数信息的 `ParameterInfo` 对象的 Python 列表。
[3]	每个 `ParameterInfo` 对象包含一个 `name` 属性，这便是参数名。在通过 SOAP 调用函数时，你不需要知道参数名，但 SOAP 支持在调用函数时使用参数名 (类似于 Python)。如果使用参数名， `WSDL.Proxy` 将会正确地把这些参数关联到远程函数。
[4]	每个参数都是都是显式类型的，使用的是在 XML Schema 定义的数据类型。你可以在上一节中发现这一点：XML Schema 命名空间是我让你忽略的模版的一部分。就目前而言，你还是可以继续忽略它。 `zipcode` 参数是一个字符串，如果你向 `WSDL.Proxy` 对象传递一个 Python 字符串，它会被正确地关联和传递到服务器。

WSDL 还允许你自省函数的返回值。

例 12.10. 揭示方法返回值

```
>>> callInfo.outparams
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AF8>]
>>> callInfo.outparams[0].name
u'return'
>>> callInfo.outparams[0].type
(u'http://www.w3.org/2001/XMLSchema', u'float')
```

[1]	与揭示函数参数的 `callInfo.inparams` 对应的是揭示返回值的 `callInfo.outparams` 。它也同样是一个列表，因为通过 SOAP 调用函数时可以返回多个值，就像 Python 函数一样。
[2]	`ParameterInfo` 对象包含 `name` 和 `type` 。这个函数返回一个浮点值，它的名字是 `return` 。

让我们整合一下，通过 WSDL proxy 调用一个 SOAP 网络服务。

例 12.11. 通过 WSDL proxy 调用一个 SOAP 网络服务

```
>>> from SOAPpy import WSDL
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl')
>>> server = WSDL.Proxy(wsdlFile)
>>> server.getTemp('90210')
66.0
>>> server.soapproxy.config.dumpSOAPOut = 1
>>> server.soapproxy.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('90210')
*** Outgoing SOAP ******************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">90210</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
*** Incoming SOAP ******************************************************
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">66.0</return>
</ns1:getTempResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
>>> temperature
66.0
```

[1]	这比直接调用 SOAP 服务时的设置简单，因为在 WSDL 文件中包含着调用服务所需要的服务 URL 和命名空间。创建 `WSDL.Proxy` 对象将会下载 WSDL 文件，解析之，并设置一个用以调用实际的 SOAP 网络服务的 `SOAPProxy` 对象。
[2]	只要创建了 `WSDL.Proxy` 对象，你就可以像调用 `SOAPProxy` 对象一样简单地调用一个函数。这并不奇怪， `WSDL.Proxy` 就是一个具有自省方法的 `SOAPProxy` 封装套件，所以调用函数的语法也是一样的。
[3]	你可以通过 `server.soapproxy` 访问 `WSDL.Proxy` 的 `SOAPProxy` 。这对于打开查错模式很重要，这样一来当你通过 WSDL proxy 调用函数时，它的 `SOAPProxy` 将会把线路上来往的 XML 文档甩下来。

12.7. 搜索 Google

让我们回到这章开始时你看到的那段代码，获得比当前气温更有价值和令人振奋的信息。

Google 提供了一个 SOAP API，以便通过程序进行 Google 搜索。使用它的前提是，你注册了 Google 网络服务。

过程 12.4. 注册 Google 网络服务

1. 访问 http://www.google.com/apis/ 并创建一个账号。唯一的需要是提供一个 E-mail 地址。注册之后，你将通过 E-mail 收到你的 Google API 许可证 (license key)。你需要在调用 Google 搜索函数时使用这个许可证。
2. 还是在 http://www.google.com/apis/ 上，下载 Google 网络 APIs 开发工具包 (Google Web APIs developer kit)。它包含着包括 Python 在内的多种语言的样例代码，更重要的是它包含着 WSDL 文件。
3. 解压这个开发工具包并找到 `GoogleSearch.wsdl` 。将这个文件拷贝到你本地驱动器的一个永久地址。在本章后面位置你会用到它。

你有了开发许可证和 Google WSDL 文件之后就可以和 Google 网络服务打交道了。

例 12.12. 内省 Google 网络服务

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl')
>>> server.methods.keys()
[u'doGoogleSearch', u'doGetCachedPage', u'doSpellingSuggestion']
>>> callInfo = server.methods['doGoogleSearch']
>>> for arg in callInfo.inparams:
... print arg.name.ljust(15), arg.type
key             (u'http://www.w3.org/2001/XMLSchema', u'string')
q               (u'http://www.w3.org/2001/XMLSchema', u'string')
start           (u'http://www.w3.org/2001/XMLSchema', u'int')
maxResults      (u'http://www.w3.org/2001/XMLSchema', u'int')
filter          (u'http://www.w3.org/2001/XMLSchema', u'boolean')
restrict        (u'http://www.w3.org/2001/XMLSchema', u'string')
safeSearch      (u'http://www.w3.org/2001/XMLSchema', u'boolean')
lr              (u'http://www.w3.org/2001/XMLSchema', u'string')
ie              (u'http://www.w3.org/2001/XMLSchema', u'string')
oe              (u'http://www.w3.org/2001/XMLSchema', u'string')
```

[1]	步入 Google 网络服务很简单：建立一个 `WSDL.Proxy` 对象并指向到你复制到本地的 Google WSDL 文件。
[2]	由 WSDL 文件可知，Google 提供三个函数： `doGoogleSearch` 、 `doGetCachedPage` 和 `doSpellingSuggestion` 。顾名思义，执行 Google 搜索并返回结果；获得 Google 最后一次扫描该页时获得的缓存；基于常见拼写错误提出单词拼写建议。
[3]	`doGoogleSearch` 函数需要一系列不同类型的参数。注意：WSDL 文件可以告诉你有哪些参数和他们的参数类型，但不能告诉你它们的含义和使用方法。在参数值有限定的情况下，理论上它能够告诉你参数的取值范围，但 Google 的 WSDL 没有那么细化。 `WSDL.Proxy` 不会变魔术，它只能给你 WSDL 文件中提供的信息。

这里简要地列出了 `doGoogleSearch` 函数的所有参数：

- `key` ——你注册 Google 网络服务时获得的 Google API 许可证。
- `q` ——你要搜索的词或词组。其语法与 Google 的网站表单处完全相同，你所知道的高级搜索语法和技巧这里完全适用。
- `start` ——起始的结果编号。与使用 Google 网页交互搜索时相同，这个函数每次返回 10 个结果。如果你需要查看 “第二” 页结果则需要将 `start` 设置为 10。
- `maxResults` ——返回的结果个数。目前的值是 10，当然如果你只对少数返回结果感兴趣或者希望节省网络带宽，也可以定义为返回更少的结果。
- `filter` ——如果设置为 `True` ，Google 将会过滤结果中重复的页面。
- `restrict` ——这里设置 `country` 并跟上一个国家代码可以限定只返回特定国家的结果。例如： `countryUK` 用于在英国搜索页面。你也可以设定 `linux` ， `mac` 或者 `bsd` 以便搜索 Google 定义的技术站点组，或者设为 `unclesam` 来搜索美国政府站点。
- `safeSearch` ——如果设置为 `True` ，Google 将会过滤掉色情站点。
- `lr` (“language restrict”，语言限制)——这里设置语言限定值返回特定语言的站点。
- `ie` 和 `oe` (“input encoding”，输入编码和 “output encoding”，输出编码)——不赞成使用，都应该是 `utf-8` 。

例 12.13. 搜索 Google

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl')
>>> key = 'YOUR_GOOGLE_API_KEY'
>>> results = server.doGoogleSearch(key, 'mark', 0, 10, False, "",
... False, "", "utf-8", "utf-8")
>>> len(results.resultElements)
10
>>> results.resultElements[0].URL
'http://diveintomark.org/'
>>> results.resultElements[0].title
'dive into <b>mark</b>'
```

[1]	在设置好 `WSDL.Proxy` 对象之后，你可以使用十个参数来调用 `server.doGoogleSearch` 。记住要使用你注册 Google 网络服务时授权给你自己的 Google API 许可证。
[2]	有很多的返回信息，但我们还是先来看一下实际的返回结果。它们被存储于 `results.resultElements` 之中，你可以像使用普通的 Python 列表那样来调用它。
[3]	`resultElements` 中的每个元素都是一个包含 `URL` 、 `title` 、 `snippet` 以及其他属性的对象。基于这一点，你可以使用诸如 **`dir(results.resultElements[0])`** 的普通 Python 自省技术来查看有效属性，或者通过 WSDL proxy 对象查看函数的 `outparams` 。不同的方法能带给你相同的结果。

`results` 对象中所加载的不仅仅是实际的搜索结果。它也含有搜索行为自身的信息，比如耗时和总结果数等 (尽管只返回了10条结果)。Google 网页界面中显示了这些信息，通过程序你也同样能获得它们。

例 12.14. 从Google获得次要信息

```
>>> results.searchTime
0.224919
>>> results.estimatedTotalResultsCount
29800000
>>> results.directoryCategories
[<SOAPpy.Types.structType item at 14367400>:
 {'fullViewableName':
  'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark',
  'specialEncoding': ''}]
>>> results.directoryCategories[0].fullViewableName
'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark'
```

[1]	这个搜索耗时 0.224919 秒。这不包括用于发送和接收 SOAP XML 文档的时间，仅仅是 Google 在接到搜索请求后执行搜索所花费的时间。
[2]	总共有接近 30,000,000 个结果信息。通过让 `start` 参数以 10 递增来重复调用 `server.doGoogleSearch` ，你能够获得全部的结果。
[3]	对于有些请求，Google 还返回一个 Google Directory 中的类别列表。你可以用这些 URLs 到 http://directory.google.com/ 建立到 directory category 页面的链接。

12.8. SOAP 网络服务故障排除

是的，SOAP 网络服务的世界中也不总是欢乐和阳光。有时候也会有故障。

正如你在本章中看到的，SOAP 牵扯了很多层面。SOAP 向 HTTP 服务器发送 XML 文档并接收返回的 XML 文档时需要用到 HTTP 层。这样一来，你在 第 11 章 *HTTP Web* 服务 学到的调试技术在这里都有了用武之地。你可以 `import httplib` 并设置 `httplib.HTTPConnection.debuglevel = 1` 来查看潜在的 HTTP 传输。

在 HTTP 层之上，还有几个可能发生问题的地方。SOAPpy 隐藏 SOAP 语法的本领令你惊叹不已，但也意味着在发生问题时更难确定问题所在。

下面的这些例子是我在使用 SOAP 网络服务时犯过的一些常见错误以及所产生的错误信息。

例 12.15. 以错误的设置调用 Proxy 方法

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> server = SOAPProxy(url)
>>> server.getTemp('27502')
<Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
```

[1]	你看出错误了吗？你手工地创建了一个 `SOAPProxy`，你正确地指定了服务 URL，但是你没有指定命名空间。由于多个服务可能被路由到相同的服务 URL，命名空间是确定你所调用的服务和方法的重要内容。
[2]	服务器返回的是一个 SOAP 错误 (Fault)，SOAPpy 把它转换为 Python 异常 `SOAPpy.Types.faultType`。从任何 SOAP 服务器返回的错误都是 SOAP 错误，因此你可以轻易地捕获这个异常。就此处而言，我们能从 SOAP 错误信息中看出端倪：由于源 `SOAPProxy` 对象没有设置服务命名空间，因此方法元素也就没有了命名空间。

错误配置 SOAP 服务的基本元素是 WSDL 着眼解决的问题。WSDL 文件包含服务 URL 和命名空间，所以你应该不会在这里犯错。但是，还有其他可能出错的地方。

例 12.16. 以错误参数调用方法

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> temperature = server.getTemp(27502)
<Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>   [2]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>
```

[1]	你看出错误了吗？这是一个不易察觉的错误：你在使用整数而不是字符串来调用 `server.getTemp` 。自省 WSDL 文件不难发现， `getTemp()` 这个 SOAP 函数接受一个参数 `zipcode` ，这是一个字符串参数。 `WSDL.Proxy` 不 会为你强制转换数据类型；你需要根据服务器需要的数据类型传递数据。
[2]	又是这样，服务器传回一个 SOAP 错误，你能从 SOAP 错误信息中看出端倪：你在使用整数类型的参数调用 `getTemp` 函数，但却没有一个以此命名的函数接收整数参数。理论上讲，SOAP 允许你重载 (*overload*) 函数，也就是可以在同一个 SOAP 服务中存在同名函数，并且参数个数也相同，但是参数的数据类型不同。这就是数据类型必须匹配的原因，也说明了为什么 `WSDL.Proxy` 不强制地为你改变数据类型。如果真的强制改变了数据类型，发生这样的错误时，调用的可能是另外一个不相干的函数。看来产生这样的错误是件幸运的事。对于数据类型多加注意会让事情简单很多，一旦搞错了数据类型便立刻会发生错误。

Python 所期待的返回值个数与远程函数的实际返回值个数不同是另一种可能的错误。

例 12.17. 调用时方法所期待的返回值个数错误

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> (city, temperature) = server.getTemp(27502)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unpack non-sequence
```

[1]	你看出错误了吗？ `server.getTemp` 只返回一个浮点值，但你写的代码却期待着获得两个值，并把它们赋值给不同的两个变量。注意这不是一个 SOAP 错误。就远程服务器而言没有发生任何错误。错误发生在完成 SOAP 交割之后。 `WSDL.Proxy` 返回一个浮点数，你本地的 Python 解释器试图将这个浮点数分成两个变量。由于函数只返回了一个值，你在试图分割它时所获得的是一个 Python 异常，而不是 SOAP 错误。

那么 Google 网络服务方面又如何呢？我曾经犯过的最常见的错误是忘记正确设置应用许可证。

例 12.18. 调用方法返回一个应用特定的错误

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy(r'/path/to/local/GoogleSearch.wsdl')
>>> results = server.doGoogleSearch('foo', 'mark', 0, 10, False, "",
... False, "", "utf-8", "utf-8")
<Fault SOAP-ENV:Server:                                              [2]
 Exception from service object: Invalid authorization key: foo:
 <SOAPpy.Types.structType detail at 14164616>:
 {'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
   at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
   at com.google.soap.search.UserKey.<init>(UserKey.java:59)
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:217)
   ... 14 more
'}>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception from service object:
Invalid authorization key: foo:
<SOAPpy.Types.structType detail at 14164616>:
{'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
```

```
    at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
    at com.google.soap.search.UserKey.<init>(UserKey.java:59)
    at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
      QueryLimits.java:217)
    ... 14 more
'}>
```

[1]	你看出错误了吗？调用的语法，参数个数以及数据类型都没有错误。这个问题是应用特定的：第一个参数应该是我的应用许可证，但 `foo` 不是一个有效的 Google 许可证。
[2]	Google 服务器返回的是一个 SOAP 错误和一大串特别长的错误信息，其中包含了完整的 Java 堆栈跟踪。记住所有 的 SOAP 错误都被标示为 SOAP Faults: errors in configuration (设置错误), errors in function arguments (函数参数错误)，或者是应用特定的错误 (这里就是) 等等。在其中埋藏的至关重要信息是： `Invalid authorization key: foo` (非有效授权许可证：foo)。

进一步阅读

- New developments for SOAPpy 一步步连接到另一个不名副其实的 SOAP 服务。

12.9. 小结

SOAP 网络服务是很复杂的，雄心勃勃的它试图涵盖网络服务的很多不同应用。这一章我们接触了它的一个简单应用。

在开始下一章的学习之前，确保你能自如地做如下工作：

- 连接到 SOAP 服务器并调用远程方法
- 通过 WSDL 文件自省远程方法
- 有效排除 SOAP 调用中的错误
- 排除常见的 SOAP 相关错误

第 13 章 单元测试

13.1. 罗马数字程序介绍 II

在前面的章节中，通过阅读代码，你迅速“深入”，以最快的速度理解了各个程序。既然你已对 Python 有了一定的了解，那么接下来让我们看看程序开发之前 的工作。

在接下来的几章中，你将会编写、调试和优化一系列工具函数来进行罗马数字和阿拉伯数字之间的转换。你已从第 7.3 节 “个案研究：罗马字母”中获知构造和验证罗马数字的机制，现在我们要做的事是退后一步去思考如何将这些机制扩展到一个双向转换的工具。

罗马数字的规则有如下一些有趣的特点：

1. 一个特定数字以罗马数字表示时只有单一方式。
2. 反之亦然：一个有效的罗马数字表示的数也只对应一个阿拉伯数字表示。(也就是说转换成阿拉伯数字表示只有一种方法。)
3. 我们研究的是 `1` 和 `3999` 之间的数字的罗马数字表示。(罗马数字有很多方法用以记录更大的数，例如在数字上加线表示 `1000` 倍的数，但你不必去理会这些。就本章而言，我们姑且把罗马数字限定在 `1` 到 `3999` 之间)。
4. 罗马数字无法表示 `0` 。(令人诧异，古罗马竟然没有 `0` 这个数字的概念。数字是为数数服务的，没有怎么数呢？)
5. 罗马数字不能表示负数。
6. 罗马数字无法表示分数和非整数。

基于如上所述，你将如何构造罗马数字转换函数呢？

`roman.py` 功能需求

1. `toRoman` 应该能返回 `1` 到 `3999` 中任意数的罗马数字表示。
2. `toRoman` 在遇到 `1` 到 `3999` 之外的数字时应该失败。
3. `toRoman` 在遇到非整数时应该失败。
4. `fromRoman` 应该能将给定的有效罗马数字表示转换为阿拉伯数字表示。
5. `fromRoman` 在遇到无效罗马数字表示时应该失败。
6. 将一个数转换为罗马数字表示，再转换回阿拉伯数字表示后应该和最初的数相同。因此， `fromRoman(toRoman(n)) == n` 对于 `1..3999` 之间所有 `n` 都适用。
7. `toRoman` 返回的罗马数字应该使用大写字母。
8. `fromRoman` 应该只接受大写罗马数字 (也就是说给定小写字母进行转换时应该失败)。

进一步阅读

- 这个站点 有关于罗马数字更多的内容，包括罗马人如何使用罗马数字的迷人 历史 (简言

之：充满偶然性和反复无常)。

13.2. 深入

现在你已经定义了你的转换程序所应有的功能，下面一步会有点儿出乎你的意料：你将要开发一个测试组件 (test suite) 来测试你未来的函数以确保它们工作正常。没错：你将为还未开发的程序开发测试代码。

这就是所谓的单元测试，因为这两个转换函数可以被当作一个单元来开发和测试，不用考虑它们可能今后成为一个大程序的一部分。Python 有一个单元测试框架，被恰如其分地称作 `unittest` 模块。

> 注意
> Python 2.1 和之后的版本已经包含了 `unittest`。Python 2.0 用户则可以从 `pyunit.sourceforge.net`下载。

单元测试是以测试为核心开发策略的重要组成部分。如果你要写单元测试代码，尽早 (最好是在被测试代码开发之前) 开发并根据代码开发和需求的变化不断更新是很重要的。单元测试不能取代更高层面的功能和系统测试，但在开发的每个阶段都很重要：

- 代码开发之前，强迫你以有效的方式考虑需求的细节。
- 代码开发中，防止过度开发。通过了所有测试用例，程序的开发就完成了。
- 重构代码时，确保新版和旧版功能一致。
- 维护代码时，当你的代码更改导致别人代码出问题时帮你留住面子。(“但是先生，我检入 (check in) 代码时所有的单元测试都通过了……”)
- 在团队开发时，可以使你有信心，保证自己提交的代码不会破坏其他人的代码，因为你可以 先运行其他人的单元测试代码。(我在“代码风暴”中见过这种事情。一个团队将任务拆分，每个人都根据自己那部分的需求开发单元测试，然后与其他成员共享。没有人会出太大的偏差而导致代码无法集成。)

13.3. `romantest.py` 介绍

这是将被开发并保存为 `roman.py` 的罗马数字转换程序的完整测试组件 (test suite)。很难立刻看出它们是如何协同工作的，似乎所有类或者方法之间都没有关系。这是有原因的，而且你很快就会明了。

例 13.1. `romantest.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
"""Unit test for roman.py"""
import roman
import unittest
class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
```

```
                      (3250, 'MMMCCL'),
                      (3313, 'MMMCCCXIII'),
                      (3408, 'MMMCDVIII'),
                      (3501, 'MMMDI'),
                      (3610, 'MMMDCX'),
                      (3743, 'MMMDCCXLIII'),
                      (3844, 'MMMDCCCXLIV'),
                      (3888, 'MMMDCCCLXXXVIII'),
                      (3940, 'MMMCMXL'),
                      (3999, 'MMMCMXCIX'))
    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
            self.assertEqual(numeral, result)
    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)
    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)
    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)
    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())
    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())
if __name__ == "__main__":
    unittest.main()
```

进一步阅读

- PyUnit 主页 对于使用 `unittest` 框架 以及本章没能涵盖的高级特性有深入的讨论。
- PyUnit FAQ 解释了 为什么测试用例要和被测试代码分开存放 。
- *Python Library Reference* 总结了 `unittest` 模块。
- ExtremeProgramming.org 讨论 你为什么需要编写单元测试。
- The Portland Pattern Repository 有一个持续的 单元测试 讨论，包括了一个 标准的定义，为什么你需要 首先开发单元测试代码 以及另外一些深层次 案例。

13.4. 正面测试 (Testing for success)

单元测试的基础是构建独立的测试用例 (test case)。一个测试用例只回答一个关于被测试代码的问题。

一个测试用例应该做到：

- 完全独立运行，不需要人工输入。单元测试应该是自动的。
- 可以自己判断被测试函数是通过还是失败，不需要人工干预结果。
- 独立运行，可以与其他测试用例隔离 (尽管它们可能测试着同一个函数)。每个测试用例是一个孤岛。

基于如上原则，让我们构建第一个测试用例。应符合如下要求：

1. `toRoman` 应该为所有 `1` 到 `3999` 的整数返回罗马数字表示。

例 13.2. `testToRomanKnownValues`

```
class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))
    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
            self.assertEqual(numeral, result)
```

[1]	编写测试用例的第一步就是继承 `unittest` 模块中的 `TestCase` 类，它提供了很多可以用在你的测试用例中来测试特定情况的有用方法。
[2]	这是我手工转换的一个 integer/numeral 对列表。它包含了最小的十个数、最大的数、每个单字符罗马数字对应的数，以及其他随机挑选的有效数样本。单元测试的关键不在于所有可能的输入，而是一个有代表性的样本。
[3]	每个独立测试本身都是一个方法，既不需要参数也不返回任何值。如果该方法正常退出没有引发异常，测试被认为通过；如果测试引发异常，测试被认为失败。
[4]	这里你真正调用了 `toRoman` 函数。(当然，函数还没有编写，但一旦被编写，这里便是调用之处。) 注意你在这里为 `toRoman` 函数定义了 API ：它必须接受整数 (待转换的数) 并返回一个字符串 (对应的罗马数字表示)，如果 API 不是这样，测试将失败。
[5]	同样值得注意，你在调用 `toRoman` 时没有试图捕捉任何可能发生的异常。这正是我们所希望的。以有效输入调用 `toRoman` 不会引发任何异常，而你看到的这些输入都是有效的。如果 `toRoman` 引发了异常，则测试失败。
[6]	假设 `toRoman` 函数被正确编写，正确调用，运行成功并返回一个值，最后一步便是检查这个返回值正确 与否。这是一个常见的问题， `TestCase` 类提供了一个方法： `assertEqual` ，来测试两个值是否相等。如果 `toRoman` 返回的结果 (`value`) 不等于我们预期的值 (`numeral`)， `assertEqual` 将会引发一个异常，测试也就此失败。如果两个值相等， `assertEqual` 什么也不做。如果每个从 `toRoman` 返回的值都等于预期值， `assertEqual` 便不会引发异常，于是 `testToRomanKnownValues` 最终正常退出，这意味着 `toRoman` 通过了该测试。

13.5. 负面测试 (Testing for failure)

使用有效输入确保函数成功通过测试还不够，你还需要测试无效输入导致函数失败的情形。但并不是任何失败都可以，必须如你预期地失败。

还记得 `toRoman` 的其他要求吧：

1. `toRoman` 在输入值为 `1` 到 `3999` 之外时失败。
2. `toRoman` 在输入值为非整数时失败。

在 Python 中，函数以引发异常的方式表示失败。`unittest` 模块提供了用于测试函数是否在给定无效输入时引发特定异常的方法。

例 13.3. 测试 `toRoman` 的无效输入

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)
    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)
    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)
    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)
```

[1]	`unittest` 模块中的 `TestCase` 类提供了 `assertRaises` 方法，它接受这几个参数：预期的异常、测试的函数，以及传递给函数的参数。(如果被测试函数有不止一个参数，把它们按顺序全部传递给 `assertRaises` ，它会把这些参数传给被测的函数。) 特别注意这里的操作：不是直接调用 `toRoman` 再手工查看是否引发特定异常 (使用 `try...except` 块捕捉异常)，`assertRaises` 为我们封装了这些。所有你要做的就是把异常 (`roman.OutOfRangeError`)、函数 (`toRoman`) 以及 `toRoman` 的参数 (`4000`) 传递给 `assertRaises` ，它会调用 `toRoman` 查看是否引发 `roman.OutOfRangeError` 异常。(还应注意到你是把 `toRoman` 函数本身当作一个参数，而不是调用它，传递它的时候也不是把它的名字作为一个字符串。我提到过吗？无论是函数还是异常， Python 中万物皆对象)。
[2]	与测试过大的数相伴的便是测试过小的数。记住，罗马数字不能表示 `0` 和负数，所以你要分别编写测试用例 (`testZero` 和 `testNegative`)。在 `testZero` 中，你测试 `toRoman` 调用 `0` 引发的 `roman.OutOfRangeError` 异常，如果没能引发 `roman.OutOfRangeError` (不论是返回了一个值还是引发了其他异常)，则测试失败。
[3]	要求 #3： `toRoman` 不能接受非整数输入，所以这里你测试 `toRoman` 在输入 `0.5` 时引发 `roman.NotIntegerError` 异常。如果 `toRoman` 没有引发 `roman.NotIntegerError` 异常，则测试失败。

接下来的两个要求与前三个类似，不同点是他们所针对的是 `fromRoman` 而不是 `toRoman` ：

1. `fromRoman` 应该能将输入的有效罗马数字转换为相应的阿拉伯数字表示。
2. `fromRoman` 在输入无效罗马数字时应该失败。

要求 #4 与要求 #1 的处理方法相同，即测试一个已知样本中的一个个数字对。要求 #5 与 #2 和 #3的处理方法相同，即通过无效输入确认 `fromRoman` 引发恰当的异常。

例 13.4. 测试 `fromRoman` 的无效输入

```
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
```

[1]	没什么新鲜的，与测试 `toRoman` 无效输入时相同的模式，只是你有了一个新的异常： `roman.InvalidRomanNumeralError` 。 `roman.py` 中一共要定义三个异常 (另外的两个是 `roman.OutOfRangeError` 和 `roman.NotIntegerError`)。稍后你在开始编写 `roman.py` 时将会知道如何定义这些异常。

13.6. 完备性检测 (Testing for sanity)

你经常会发现一组代码中包含互逆的转换函数，一个把 A 转换为 B ，另一个把 B 转换为 A。在这种情况下，创建“完备性检测”可以使你在由 A 转 B 再转 A 的过程中不会出现丢失精度或取整等错误。

考虑这个要求：

1. 如果你给定一个数，把它转化为罗马数字表示，然后再转换回阿拉伯数字表示，你所得到的应该是最初给定的那个数。因此，对于 `1..3999` 中的 `n`， `fromRoman(toRoman(n)) == n` 总成立。

例 13.5. 以 `toRoman` 测试 `fromRoman` 的输出

```
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)
```

[1]	你已经见到过 `range` 函数，但这里它以两个参数被调用，返回了从第一个参数 (`1`) 开始到但不包括 第二个参数 (`4000`) 的整数列表。因此， `1..3999` 就是准备转换为罗马数字表示的有效值列表。
[2]	我想提一下，这里的 `integer` 并不是一个 Python 关键字，而只是没有什么特别的变量名。
[3]	这里的测试逻辑显而易见：把一个数 (`integer`) 转换为罗马数字表示的数 (`numeral`)，然后再转换回来 (`result`) 并确保最后的结果和最初的数是同一个数。如果不是， `assertEqual` 便会引发异常，测试也便立刻失败。如果所有的结果都和初始数一致， `assertEqual` 将会保持沉默，整个 `testSanity` 方法将会最终也保持沉默，测试则将会被认定为通过。

最后两个要求和其他的要求不同，似乎既武断而又微不足道：

1. `toRoman` 返回的罗马数字应该使用大写字母。
2. `fromRoman` 应该只接受大写罗马数字 (也就是说给定小写字母进行转换时应该失败)。

事实上，它们确实有点武断，譬如你完全可以让 `fromRoman` 接受小写和大小写混合的输入；但他们也不是完全武断；如果 `toRoman` 总是返回大写的输出，那么 `fromRoman` 至少应该接受大写字母输入，不然 “完备性检测” (要求 #6) 就会失败。不管怎么说，只 接受大写输入还是武断的，但就像每个系统都会告诉你的那样，大小写总会出问题，因此事先规定这一点还是有必要的。既然有必要规定，那么也就有必要测试。

例 13.6. 大小写测试

```
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())
    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())
```

[1]	关于这个测试用例最有趣的一点不在于它测试了什么，而是它不测试什么。它不会测试 `toRoman` 的返回值是否正确或者一致；这些问题由其他测试用例来回答。整个测试用例仅仅测试大写问题。你也许觉得应该将它并入到完备性测试，毕竟都要遍历整个输入值范围并调用 `toRoman`。[11]但是这样将会违背一条基本规则：每个测试用例只回答一个的问题。试想一下，你将这个测试并入到完备性测试中，然后遇到了测试失败。你还需要进一步分析以便判定测试用例的哪部分出了问题。如果你需要分析方能找出问题所在，无疑你的测试用例在设计上出了问题。
[2]	这有一个和前面相似的情况：尽管 "你知道" `toRoman` 总是返回大写字母，你还是需要把返回值显式地转换成大写字母后再传递给只接受大写的 `fromRoman` 进行测试。为什么？因为 `toRoman` 只返回大写字母是一个独立的需求。如果你改变了这个需求，例如改成总是返回小写字母，那么 `testToRomanCase` 测试用例也应作出调整，但这个测试用例应该仍能通过。这是另外一个基本规则：每个测试用例必须可以与其他测试用例隔离工作，每个测试用例是一个"孤岛"。
[3]	注意你并没有使用 `fromRoman` 的返回值。这是一个有效的 Python 语法：如果一个函数返回一个值，但没有被使用，Python 会直接把这个返回值扔掉。这正是你所希望的，这个测试用例并不对返回值进行测试，只是测试 `fromRoman` 接受大写字母而不引发异常。
[4]	这行有点复杂，但是它与 `ToRomanBadInput` 和 `FromRomanBadInput` 测试很相似。你在测试以特定值 (`numeral.lower()` ，循环中目前罗马数字的小写版) 调用特定函数 (`roman.fromRoman`) 会确实引发特定的异常 (`roman.InvalidRomanNumeralError`)。如果 (在循环中的每一次) 确实如此，测试通过；如果有一次不是这样 (比如引发另外的异常或者不引发异常)，测试失败。

在下一章中，你将看到如何编写可以通过这些测试的代码。

Footnotes

[11] "除了诱惑什么我都能抗拒。 (I can resist everything except temptation.)"——Oscar Wilde

第 14 章 测试优先编程

14.1. `roman.py` , 第 1 阶段

到目前为止，单元测试已经完成，是时候开始编写被单元测试测试的代码了。你将分阶段地完成这个工作，因此开始时所有的单元测试都是失败的，但在逐步完成 `roman.py` 的同时你会看到它们一个个地通过测试。

例 14.1. `roman1.py`

这个程序可以在例子目录下的 `py/roman/stage1/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
"""Convert to and from Roman numerals"""
#Define exceptions
class RomanError(Exception): pass                     class OutOfRangeError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
def toRoman(n):
    """convert integer to Roman numeral"""
    pass
def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

[1]	这就是如何定义你自己的 Python 异常。异常 (Exception) 也是类，通过继承已有的异常，你可以创建自定义的异常。强烈建议 (但不是必须) 你继承 `Exception` 来定义自己的异常，因为它是所有内建异常的基类。这里我定义了 `RomanError` (从 `Exception` 继承而来) 作为我所有自定义异常的基类。这是一个风格问题，我也可以直接从 `Exception` 继承建立每一个自定义异常。
[2]	`OutOfRangeError` 和 `NotIntegerError` 异常将会最终被用于 `toRoman` 以标示不同类型的无效输入，更具体而言就是 `ToRomanBadInput` 测试的那些。
[3]	`InvalidRomanNumeralError` 将被最终用于 `fromRoman` 以标示无效输入，具体而言就是 `FromRomanBadInput` 测试的那些。
[4]	在这一步中你只是想定义每个函数的 API ，而不想具体实现它们，因此你以 Python 关键字 `pass` 姑且带过。

重要的时刻到了 (请打起鼓来)：你终于要对这个简陋的小模块开始运行单元测试了。目前而言，每一个测试用例都应该失败。事实上，任何测试用例在此时通过，你都应该回头看看 `romantest.py` ，仔细想想为什么你写的测试代码如此没用，以至于连什么都不作的函数都能通过测试。

用命令行选项 `-v` 运行 `romantest1.py` 可以得到更详细的输出信息，这样你就可以看到每一个测试用例的具体运行情况。如果幸运，你的结果应该是这样的：

例 14.2. 以 `romantest1.py` 测试 `roman1.py` 的输出

```
fromRoman should only accept uppercase input ... ERROR
toRoman should always return uppercase ... ERROR
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... FAIL
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL
======================================================================
ERROR: fromRoman should only accept uppercase input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 154, in testFromRomanCase
    roman1.fromRoman(numeral.upper())
AttributeError: 'None' object has no attribute 'upper' ==================================
ERROR: toRoman should always return uppercase
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 148, in testToRomanCase
    self.assertEqual(numeral, numeral.upper())
AttributeError: 'None' object has no attribute 'upper' ==================================
FAIL: fromRoman should fail with malformed antecedents
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 133, in testMalformedAntecede
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with repeated pairs of numerals
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 127, in testRepeatedPairs
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with too many repeated numerals
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 122, in testTooManyRepeatedNu
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should give known result with known input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 99, in testFromRomanKnownValu
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None ===============================================================
FAIL: toRoman should give known result with known input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 93, in testToRomanKnownValues
    self.assertEqual(numeral, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: I != None ===============================================================
FAIL: fromRoman(toRoman(n))==n for all n
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None ===============================================================
FAIL: toRoman should fail with non-integer input
---------------------------------------------------------------------- Traceback (most re
```

```
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 116, in testNonInteger
    self.assertRaises(roman1.NotIntegerError, roman1.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError ==========================================================
FAIL: toRoman should fail with negative input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 112, in testNegative
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError ==========================================================
FAIL: toRoman should fail with large input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 104, in testTooLarge
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError ==========================================================
FAIL: toRoman should fail with 0 input  --------------------------------------------------
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 108, in testZero
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError  -------------------------------------------------------
Ran 12 tests in 0.040s  FAILED (failures=10, errors=2)
```

[1]	运行脚本将会执行 `unittest.main()`，由它来执行每个测试用例，也就是每个在 `romantest.py` 中定义的方法。对于每个测试用例，无论测试通过与否，都会输出这个方法的 `doc string`。意料之中，没有通过一个测试用例。
[2]	对于每个失败的测试用例，`unittest` 显示的跟踪信息告诉我们都发生了什么。就此处而言，调用 `assertRaises` (也称作 `failUnlessRaises`) 引发了一个 `AssertionError` 异常，因为期待 `toRoman` 所引发的 `OutOfRangeError` 异常没有出现。
[3]	在这些细节后面，`unittest` 给出了一个关于被执行测试的个数和花费时间的总结。
[4]	总而言之，由于至少一个测试用例没有通过，单元测试失败了。当某个测试用例没能通过时，`unittest` 会区分是失败 (failures) 还是错误 (errors)。失败是指调用 `assertXYZ` 方法，比如 `assertEqual` 或者 `assertRaises` 时，断言的情况没有发生或预期的异常没有被引发。而错误是指你测试的代码或单元测试本身发生了某种异常。例如：`testFromRomanCase` 方法 (“ `fromRoman` 只接受大写输入”) 就是一个错误，因为调用 `numeral.upper()` 引发了一个 `AttributeError` 异常，因为 `toRoman` 的返回值不是期望的字符串类型。但是，`testZero` (“ `toRoman` 应该在输入 0 时失败”) 是一个失败，因为调用 `fromRoman` 没有引发一个 `assertRaises` 期待的异常：`InvalidRomanNumeral`。

14.2. `roman.py` , 第 2 阶段

现在你有了 `roman` 模块的大概框架，到了开始写代码以通过测试的时候了。

例 14.3. `roman2.py`

这个文件可以从 `py/roman/stage2/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
"""Convert to and from Roman numerals"""
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))
def toRoman(n):
    """convert integer to Roman numeral"""
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result
def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

[1]	`romanNumeralMap` 是一个用来定义三个内容的元组的元组：1. 代表大部分罗马数字的字符。注意不只是单字符的罗马数字，你同样在这里定义诸如 `CM` (“比一千少一百，即 900”) 的双字符，这可以让稍后编写的 `toRoman` 简单一些。2. 罗马数字的顺序。它们是以降序排列的，从 `M` 一路到 `I` 。3. 每个罗马数字所对应的数值。每个内部的元组都是一个 `(_numeral_, _value_)` 数值对。
[2]	这里便显示出你丰富的数据结构带来的优势，你不需要什么特定的逻辑处理减法规则。你只需要通过搜寻 `romanNumeralMap` 寻找不大于输入数值的最大对应整数即可。只要找到，就在结果的结尾把这个整数对应的罗马字符添加到输出结果的末尾，从输入值中减去这个整数，一遍遍这样继续下去。

例 14.4. `toRoman` 如何工作

如果你不明了 `toRoman` 如何工作，在 `while` 循环的结尾添加一个 `print` 语句：

```
        while n >= integer:
            result += numeral
            n -= integer
            print 'subtracting', integer, 'from input, adding', numeral, 'to output'
```

```
>>> import roman2
>>> roman2.toRoman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

看来 `toRoman` 可以运转了，至少手工测试可以。但能通过单元测试吗？啊哈，不，不完全可以。

例 14.5. 以 `romantest2.py` 测试 `roman2.py` 的输出

要记得用 `-v` 命令行选项运行 `romantest2.py` 开启详细信息模式。

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok  fromRoman should fail with malformed antec
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok  fromRoman(toRoman(n))==n for al
toRoman should fail with non-integer input ... FAIL  toRoman should fail with negative in
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL
```

[1]	事实上，`toRoman` 的返回值总是大写的，因为 `romanNumeralMap` 定义的罗马字符都是以大写字母表示的。因此这个测试已经通过了。
[2]	好消息来了：这个版本的 `toRoman` 函数能够通过已知值测试。记住，这并不能证明完全没问题，但至少通过测试多种有效输入考验了这个函数：包括每个单一字符的罗马数字，可能的最大输入 (`3999`)，以及可能的最长的罗马数字 (对应于 `3888`)。从这点来看，你有理由相信这个函数对于任何有效输入都不会出问题。
[3]	但是，函数还没办法处理无效输入，每个无效输入测试都失败了。这很好理解，因为你还没有对无效输入进行检查，测试用例希望捕捉到特定的异常 (通过 `assertRaises`)，而你根本没有让这些异常引发。这是你下一阶段的工作。

下面是单元测试结果的剩余部分，列出了所有失败的详细信息，你已经让它降到了 10 个。

```
 ======================================================================
FAIL: fromRoman should only accept uppercase input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 156, in testFromRomanCase
    roman2.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with malformed antecedents
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 133, in testMalformedAntecede
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with repeated pairs of numerals
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 127, in testRepeatedPairs
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with too many repeated numerals
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 122, in testTooManyRepeatedNu
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should give known result with known input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 99, in testFromRomanKnownValu
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None ===============================================================
FAIL: fromRoman(toRoman(n))==n for all n
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None ===============================================================
FAIL: toRoman should fail with non-integer input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 116, in testNonInteger
    self.assertRaises(roman2.NotIntegerError, roman2.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError =========================================================
FAIL: toRoman should fail with negative input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 112, in testNegative
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError =========================================================
FAIL: toRoman should fail with large input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 104, in testTooLarge
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError =========================================================
FAIL: toRoman should fail with 0 input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 108, in testZero
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError ---------------------------------------------------------
```

```
Ran 12 tests in 0.320s
FAILED (failures=10)
```

14.3. `roman.py` , 第 3 阶段

现在 `toRoman` 对于有效的输入 (`1` 到 `3999` 整数) 已能正确工作，是正确处理那些无效输入 (任何其他输入) 的时候了。

例 14.6. `roman3.py`

这个文件可以在例子目录下的 `py/roman/stage3/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
"""Convert to and from Roman numerals"""
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))
def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result
def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

[1]	这个写法很 Pythonic：一次进行多个比较。这等价于 `if not ((0 < n) and (n < 4000))`，但是更容易让人理解。这是在进行范围检查，可以将过大的数、负数和零查出来。
[2]	你使用 `raise` 语句引发自己的异常。你可以引发任何内建异常或者已定义的自定义异常。第二个参数是可选的，如果给定，则会在异常未被处理时显示于追踪信息 (trackback) 之中。
[3]	这是一个非整数检查。非整数无法转化为罗马数字表示。
[4]	函数的其他部分未被更改。

例 14.7. 观察 `toRoman` 如何处理无效输入

```
>>> import roman3
>>> roman3.toRoman(4000)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 27, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
>>> roman3.toRoman(1.5)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 29, in toRoman
    raise NotIntegerError, "non-integers can not be converted"
NotIntegerError: non-integers can not be converted
```

例 14.8. 用 `romantest3.py` 测试 `roman3.py` 的结果

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok  fromRoman(toRoman(n))==n for al
toRoman should fail with non-integer input ... ok  toRoman should fail with negative inpu
toRoman should fail with 0 input ... ok
```

[1]	`toRoman` 仍然能通过已知值测试，这很令人鼓舞。所有第 2 阶段通过的测试仍然能通过，这说明新的代码没有对原有代码构成任何负面影响。
[2]	更令人振奋的是所有的无效输入测试现在都通过了。 `testNonInteger` 这个测试能够通过是因为有了 `int(n) <> n` 检查。当一个非整数传递给 `toRoman` 时， `int(n) <> n` 检查出问题并引发 `NotIntegerError` 异常，这正是 `testNonInteger` 所期待的。
[3]	`testNegative` 这个测试能够通过是因为 `not (0 < n < 4000)` 检查引发了 `testNegative` 期待的 `OutOfRangeError` 异常。

```
 ======================================================================
FAIL: fromRoman should only accept uppercase input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 156, in testFromRomanCase
    roman3.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with malformed antecedents
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 133, in testMalformedAntecede
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with repeated pairs of numerals
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 127, in testRepeatedPairs
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with too many repeated numerals
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 122, in testTooManyRepeatedNu
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should give known result with known input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 99, in testFromRomanKnownValu
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None ==============================================================
FAIL: fromRoman(toRoman(n))==n for all n
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None --------------------------------------------------------------
Ran 12 tests in 0.401s
FAILED (failures=6)
```

[1]	你已将失败降至 6 个，而且它们都是关于 `fromRoman` 的：已知值测试、三个独立的无效输入测试，大小写检查和完备性检查。这意味着 `toRoman` 通过了所有可以独立通过的测试 (完备性测试也测试它，但需要 `fromRoman` 编写后一起测试)。这就是说，你应该停止对 `toRoman` 的代码编写。不必再推敲，不必再做额外的检查“恰到好处”。停下来吧！现在，别再敲键盘了。

> 注意
>
> 全面的单元测试能够告诉你的最重要的事情是什么时候停止编写代码。当一个函数的所有单元测试都通过了，停止编写这个函数。一旦整个模块的单元测试通过了，停止编写这个模块。

14.4. `roman.py`, 第 4 阶段

现在 `toRoman` 完成了，是开始编写 `fromRoman` 的时候了。感谢那个将每个罗马数字和对应整数关连的完美数据结构，这个工作不比 `toRoman` 函数复杂。

例 14.9. `roman4.py`

这个文件可以在例子目录下的 `py/roman/stage4/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
"""Convert to and from Roman numerals"""
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))
# toRoman function omitted for clarity (it hasn't changed)
def fromRoman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

[1]	这和 `toRoman` 的工作模式很相似。你遍历整个罗马数字数据结构 (一个元组的元组)，与前面不同的是不去一个个搜寻最大的整数，而是搜寻 “最大的”罗马数字字符串。

例 14.10. `fromRoman` 如何工作

如果你不清楚 `fromRoman` 如何工作，在 `while` 结尾处添加一个 `print` 语句：

```
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print 'found', numeral, 'of length', len(numeral), ', adding', integer
```

```
>>> import roman4
>>> roman4.fromRoman('MCMLXXII')
found M , of length 1, adding 1000
found CM , of length 2, adding 900
found L , of length 1, adding 50
found X , of length 1, adding 10
found X , of length 1, adding 10
found I , of length 1, adding 1
found I , of length 1, adding 1
1972
```

例 14.11. 用 `romantest4.py` 测试 `roman4.py` 的结果

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... ok  toRoman should give known res
fromRoman(toRoman(n))==n for all n ... ok  toRoman should fail with non-integer input ...
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

[1]	这儿有两个令人激动的消息。一个是 `fromRoman` 对于所有有效输入运转正常，至少对于你测试的已知值是这样。
[2]	第二个好消息是，完备性测试也通过了。与已知值测试的通过一起来看，你有理由相信 `toRoman` 和 `fromRoman` 对于所有有效输入值工作正常。(尚不能完全相信，理论上存在这种可能性： `toRoman` 存在错误而导致一些特定输入会产生错误的罗马数字表示，并且 `fromRoman` 也存在相应的错误，把 `toRoman` 错误产生的这些罗马数字错误地转换为最初的整数。取决于你的应用程序和你的要求，你或许需要考虑这个可能性。如果是这样，编写更全面的测试用例直到解决这个问题。)

```
 ======================================================================
FAIL: fromRoman should only accept uppercase input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 156, in testFromRomanCase
    roman4.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with malformed antecedents
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 133, in testMalformedAntecede
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with repeated pairs of numerals
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 127, in testRepeatedPairs
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ================================================
FAIL: fromRoman should fail with too many repeated numerals
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 122, in testTooManyRepeatedNu
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ------------------------------------------------
Ran 12 tests in 1.222s
FAILED (failures=4)
```

14.5. `roman.py` , 第 5 阶段

现在 `fromRoman` 对于有效输入能够正常工作了，是揭开最后一个谜底的时候了：使它正常工作于无效输入的情况下。这意味着要找出一个方法检查一个字符串是不是有效的罗马数字。这比 `toRoman` 中验证有效的数字输入困难，但是你可以使用一个强大的工具：正则表达式。

如果你不熟悉正则表达式，并且没有读过 第 7 章 正则表达式，现在是该好好读读的时候了。

如你在 第 7.3 节 “个案研究：罗马字母”中所见到的，构建罗马数字有几个简单的规则：使用字母 `M` , `D` , `C` , `L` , `X` , `V` 和 `I` 。让我们回顾一下：

1. 字符是被“加”在一起的：`I` 是 `1`， `II` 是 `2`， `III` 是 `3`。 `VI` 是 `6` (看上去就是 “ `5` 加 `1` ”)， `VII` 是 `7`， `VIII` 是 `8` 。
2. 这些字符 (`I` , `X` , `C` 和 `M`) 最多可以重复三次。对于 `4`，你则需要利用下一个能够被5整除的字符进行减操作得到。你不能把 `4` 表示为 `IIII` 而应该表示为 `IV` (“比 `5` 小 `1` ”)。 `40` 则被写作 `XL` (“比 `50` 小 `10` ”)， `41` 表示为 `XLI` ， `42` 表示为 `XLII` ， `43` 表示为 `XLIII` ， `44` 表示为 `XLIV` (“比 `50` 小 `10` ，加上 `5` 小 `1` ”)。
3. 类似地，对于数字 `9` ，你必须利用下一个能够被10整除的字符进行减操作得到：`8` 是 `VIII` ，而 `9` 是 `IX` (“比 `10` 小 `1` ”)，而不是 `VIIII` (由于 `I` 不能重复四次)。 `90` 表示为 `XC` ， `900` 表示为 `CM` 。
4. 含五的字符不能被重复：`10` 应该表示为 `X` ，而不会是 `VV` 。 `100` 应该表示为 `C` ，而不是 `LL` 。
5. 罗马数字一般从高位到低位书写，从左到右阅读，因此不同顺序的字符意义大不相同。 `DC` 是 `600` ， `CD` 是完全另外一个数 (`400` ，“比 `500` 少 `100` ”)。 `CI` 是 `101` ，而 `IC` 根本就不是一个有效的罗马数字 (因为你无法从 `100` 直接减 `1` ，应该写成 `XCIX` ，意思是 “比 `100` 少 `10` ，然后加上数字 `9` ，也就是比 `10` 少 `1` ”)。

例 14.12. `roman5.py`

这个程序可以在例子目录下的 `py/roman/stage5/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
"""Convert to and from Roman numerals"""
import re
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))
def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result
#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

[1]	这只是 第 7.3 节 “个案研究：罗马字母” 中讨论的匹配模版的继续。十位上可能是 `XC` (`90`)，`XL` (`40`)，或者可能是 `L` 后面跟着 0 到 3 个 `X` 字符。个位则可能是 `IX` (`9`)，`IV` (`4`)，或者是一个可能是 `V` 后面跟着 0 到 3 个 `I` 字符。
[2]	把所有的逻辑编码成正则表达式，检查无效罗马字符的代码就很简单了。如果 `re.search` 返回一个对象则表示匹配了正则表达式，输入是有效的，否则输入无效。

这里你可能会怀疑，这个面目可憎的正则表达式是否真能查出错误的罗马字符表示。没关系，不必完全听我的，不妨看看下面的结果：

例 14.13. 用 `romantest5.py` 测试 `roman5.py` 的结果

```
 fromRoman should only accept uppercase input ... ok  toRoman should always return upperc
fromRoman should fail with malformed antecedents ... ok  fromRoman should fail with repea
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
----------------------------------------------------------------------
Ran 12 tests in 2.864s
OK
```

[1]	有件事我未曾讲过，那就是默认情况下正则表达式大小写敏感。由于正则表达式 `romanNumeralPattern` 是以大写字母构造的， `re.search` 将拒绝不全部是大写字母构成的输入。因此大写输入的检查就通过了。
[2]	更重要的是，无效输入测试也通过了。例如，上面这个用例测试了 `MCMC` 之类的情形。正如你所见，这不匹配正则表达式，因此 `fromRoman` 引发一个测试用例正在等待的 `InvalidRomanNumeralError` 异常，所以测试通过了。
[3]	事实上，所有的无效输入测试都通过了。正则表达式捕捉了你在编写测试用例时所能预见的所有情况。
[4]	最终迎来了“ `OK` ”这个平淡的“年度大奖”，所有测试都通过后 `unittest` 模块就会输出它。

> 注意
>
> 当所有测试都通过了，停止编程。

第 15 章 重构

15.1. 处理 bugs

尽管你很努力地编写全面的单元测试，但是 bug 还是会出现。我所说的 “bug” 是什么呢？Bug 是你还没有编写的测试用例。

例 15.1. 关于 Bug

```
>>> import roman5
>>> roman5.fromRoman("")
0
```

[1]	在前面的章节中你注意到一个空字符串会匹配上那个检查罗马数字有效性的正则表达式了吗？对于最终版本中的正则表达式这一点仍然没有改变。这就是一个 Bug，你希望空字符串能够像其他无效的罗马数字表示一样引发 `InvalidRomanNumeralError` 异常。

在重现这个 Bug 并修改它之前你应该编写一个会失败的测试用例来说明它。

例 15.2. 测试 bug (`romantest61.py`)

```
class FromRomanBadInput(unittest.TestCase):
    # previous test cases omitted for clarity (they haven't changed)
    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, "")
```

[1]	这里很简单。以空字符串调用 `fromRoman` 并确保它会引发一个 `InvalidRomanNumeralError` 异常。难点在于找出 Bug，既然你已经知道它了，测试就简单了。

因为你的代码存在一个 Bug，并且你编写了测试这个 Bug 的测试用例，所以测试用例将会失败：

例 15.3. 用 `romantest61.py` 测试 `roman61.py` 的结果

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... FAIL
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
======================================================================
FAIL: fromRoman should fail with blank string
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage6\romantest61.py", line 137, in testBlank
    self.assertRaises(roman61.InvalidRomanNumeralError, roman61.fromRoman, "")
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError ------------------------------------------------
Ran 13 tests in 2.864s
FAILED (failures=1)
```

现在 你可以修改这个 Bug了。

例 15.4. 修改 Bug (`roman62.py`)

这个文件可以在例子目录下的 `py/roman/stage6/` 目录中找到。

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

[1]	只需要两行代码：一行直接检查空字符串和一行 `raise` 语句。

例 15.5. 用 `romantest62.py` 测试 `roman62.py` 的结果

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok  fromRoman should fail with malformed ante
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
----------------------------------------------------------------------
Ran 13 tests in 2.834s
OK
```

[1]	空字符串测试用例现在通过了，说明 Bug 被修正了。
[2]	所有其他测试用例依然通过，证明这个 Bug 修正没有影响到其他部分。不需要再编程了。

这样编程，并没有令 Bug 修正变得简单。简单的 Bug (就像这一个) 需要简单的测试用例，复杂 Bug 则需要复杂的测试用例。以测试为核心的氛围好像 延长了修正 Bug 的时间，因为你需要先贴切地描述出 Bug (编写测试用例) 然后才去修正它。如果测试用例没能正确通过，你需要思量这个修改错了还是测试用例本身出现了 Bug。无论如何，从长远上讲，这样在测试代码和代码之间的反复是值得的，因为这样会使 Bug 在第一时间就被修正的可能性大大提高。而且不论如何更改，你都可以轻易地重新运行所有 测试用例，新代码破坏老代码的机会也变得微乎其微。今天的单元测试就是明天的回归测试 (regression test)。

15.2. 应对需求变化

尽管你竭尽努力地分析你的客户，并点灯熬油地提炼出精确的需求，但需求还是会是不断变化。大部分客户在看到产品前不知道他们想要什么。即便知道，也不擅于精确表述出他们的有效需求。即便能表述出来，他们在下一个版本一定会要求更多的功能。因此你需要做好更新测试用例的准备以应对需求的改变。

假设你想要扩展罗马数字转换函数的范围。还记得没有哪个字符可以重复三遍以上这条规则吗？呃，现在罗马人希望给这条规则来个例外，用连续出现 4 个 `M` 字符来表示 `4000`。如果这样改了，你就可以把转换范围从 `1..3999` 扩展到 `1..4999`。但你先要对测试用例进行修改。

例 15.6. 修改测试用例以适应新需求 (`romantest71.py`)

这个文件可以在例子目录下的 `py/roman/stage7/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
 import roman71
import unittest
class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
```

```
                      (1832, 'MDCCCXXXII'),
                      (1993, 'MCMXCIII'),
                      (2074, 'MMLXXIV'),
                      (2152, 'MMCLII'),
                      (2212, 'MMCCXII'),
                      (2343, 'MMCCCXLIII'),
                      (2499, 'MMCDXCIX'),
                      (2574, 'MMDLXXIV'),
                      (2646, 'MMDCXLVI'),
                      (2723, 'MMDCCXXIII'),
                      (2892, 'MMDCCCXCII'),
                      (2975, 'MMCMLXXV'),
                      (3051, 'MMMLI'),
                      (3185, 'MMMCLXXXV'),
                      (3250, 'MMMCCL'),
                      (3313, 'MMMCCCXIII'),
                      (3408, 'MMMCDVIII'),
                      (3501, 'MMMDI'),
                      (3610, 'MMMDCX'),
                      (3743, 'MMMDCCXLIII'),
                      (3844, 'MMMDCCCXLIV'),
                      (3888, 'MMMDCCCLXXXVIII'),
                      (3940, 'MMMCMXL'),
                      (3999, 'MMMCMXCIX'),
                      (4000, 'MMMM'),
                      (4500, 'MMMMD'),
                      (4888, 'MMMMDCCCLXXXVIII'),
                      (4999, 'MMMMCMXCIX'))
    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.toRoman(integer)
            self.assertEqual(numeral, result)
    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 5000)
    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 0)
    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, -1)
    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman71.NotIntegerError, roman71.toRoman, 0.5)
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)
    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)
    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)
    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, "")
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 5000):
```

```
            numeral = roman71.toRoman(integer)
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())
    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            roman71.fromRoman(numeral.upper())
            self.assertRaises(roman71.InvalidRomanNumeralError,
                              roman71.fromRoman, numeral.lower())
if __name__ == "__main__":
    unittest.main()
```

[1]	原来的已知值没有改变 (它们仍然是合理的测试值) 但你需要添加几个大于 `4000` 的值。这里我添加了 `4000` (最短的一个), `4500` (次短的一个), `4888` (最长的一个) 和 `4999` (值最大的一个)。
[2]	“最大输入”的定义改变了。以前是以 `4000` 调用 `toRoman` 并期待一个错误；而现在 `4000-4999` 成为了有效输入，需要将这个最大输入提升至 `5000` 。
[3]	“过多字符重复” 的定义也改变了。这个测试以前是以 `'MMMM'` 调用 `fromRoman` 并期待一个错误；而现在 `MMMM` 被认为是一个有效的罗马数字表示，需要将这个“过多字符重复”改为 `'MMMMM'` 。
[4]	完备测试和大小写测试原来在 `1` 到 `3999` 范围内循环。现在范围扩展了，这个 `for` 循环需要将范围也提升至 `4999` 。

现在你的测试用例和新需求保持一致了，但是你的程序代码还没有，因此几个测试用例的失败是意料之中的事。

例 15.7. 用 `romantest71.py` 测试 `roman71.py` 的结果

```
 fromRoman should only accept uppercase input ... ERROR  toRoman should always return upp
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ERROR  toRoman should give known
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

[1]	我们的大小写检查是因为循环范围是 `1` 到 `4999`，而 `toRoman` 只接受 `1` 到 `3999` 之间的数，因此测试循环到 `4000` 就会失败。
[2]	`fromRoman` 的已知值测试在遇到 `'MMMM'` 就会失败，因为 `fromRoman` 还认为这是一个无效的罗马数字表示。
[3]	`toRoman` 的已知值测试在遇到 `4000` 就会失败，因为 `toRoman` 仍旧认为这超出了有效值范围。
[4]	完备测试在遇到 `4000` 也会失败，因为 `toRoman` 也会认为这超出了有效值范围。

```
 ======================================================================
ERROR: fromRoman should only accept uppercase input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 161, in testFromRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999) ===================================
ERROR: toRoman should always return uppercase
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 155, in testToRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999) ===================================
ERROR: fromRoman should give known result with known input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 102, in testFromRomanKnownVa
    result = roman71.fromRoman(numeral)
  File "roman71.py", line 47, in fromRoman
    raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
InvalidRomanNumeralError: Invalid Roman numeral: MMMM ====================================
ERROR: toRoman should give known result with known input
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 96, in testToRomanKnownValue
    result = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999) ===================================
ERROR: fromRoman(toRoman(n))==n for all n
---------------------------------------------------------------------- Traceback (most re
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 147, in testSanity
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999) -----------------------------------
Ran 13 tests in 2.213s
FAILED (errors=5)
```

既然新的需求导致了测试用例的失败，你该考虑修改代码以便它能再次通过测试用例。(在你开始编写单元测试时要习惯一件事：被测试代码永远不会在编写测试用例“之前”编写。正因为如此，你还有一些工作要做，一旦可以通过所有的测试用例，停止编码。)

例 15.8. 为新的需求编写代码 (`roman72.py`)

这个文件可以在例子目录下的 `py/roman/stage7/` 目录中找到。

```
"""Convert to and from Roman numerals"""
import re
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))
def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 5000):
        raise OutOfRangeError, "number out of range (must be 1..4999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result
#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

[1]	toRoman 只需要在取值范围检查一处做个小改动。将原来的 0 < n < 4000 ，更改为现在的检查 0 < n < 5000 。你还要更改你 raise 的错误信息以反映接受新取值范围 (1..4999 而不再是 1..3999)。你不需要改变函数的其他部分，它们已经适用于新的情况。(它们会欣然地为新的 1000 添加 'M' ，以 4000 为例，函数会返回 'MMMM' 。之前没能这样做是因为到范围检查时就被停了下来。)
[2]	你对 fromRoman 也不需要做过多的修改。唯一的修改就在 romanNumeralPattern ：如果你注意的话，你会发现你只需在正则表达式的第一部分增加一个可选的 M 。这就允许最多 4 个 M 字符而不再是 3 个，意味着你允许代表 4999 而不只是 3999 的罗马数字。 fromRoman 函数本身是普遍适用的，它并不在意字符被多少次的重复，只是根据重复的罗马字符对应的数值进行累加。以前没能处理 'MMMM' 是因为你通过正则表达式的检查强行停止了。

你可能会怀疑只需这两处小改动。嘿，不相信我的话，你自己看看吧：

例 15.9. 用 `romantest72.py` 测试 `roman72.py` 的结果

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
----------------------------------------------------------------------
Ran 13 tests in 3.685s
OK
```

[1]	所有的测试用例都通过了，停止编写代码。

全面的单元测试意味着不必依赖于程序员的一面之词：“相信我！”

15.3. 重构

全面的单元测试带来的最大好处不是你的全部测试用例最终通过时的成就感；也不是被责怪破坏了别人的代码时能够证明 自己的自信。最大的好处是单元测试给了你自由去无情地重构。

重构是在可运行代码的基础上使之工作得更好的过程。通常，“更好”意味着“更快”，也可能意味着 “使用更少的内存”，或者 “使用更少的磁盘空间”，或者仅仅是“更优雅的代码”。不管对你，对你的项目意味什么，在你的环境中，重构对任何程序的长期良性运转都是重要的。

这里，“更好” 意味着 “更快”。更具体地说，`fromRoman` 函数可以更快，关键在于那个丑陋的、用于验证罗马数字有效性的正则表达式。尝试不用正则表达式去解决是不值得的 (这样做很难，而且可能也快不了多少)，但可以通过预编译正则表达式使函数提速。

例 15.10. 编译正则表达式

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')
<SRE_Match object at 01090490>
>>> compiledPattern = re.compile(pattern)
>>> compiledPattern
<SRE_Pattern object at 00F06E28>
>>> dir(compiledPattern)
['findall', 'match', 'scanner', 'search', 'split', 'sub', 'subn']
>>> compiledPattern.search('M')
<SRE_Match object at 01104928>
```

[1]	这是你看到过的 `re.search` 语法。把一个正则表达式作为字符串 (`pattern`) 并用这个字符串来匹配 (`'M'`)。如果能够匹配，函数返回 一个 match 对象，可以用来确定匹配的部分和如何匹配的。
[2]	这里是一个新的语法： `re.compile` 把一个正则表达式作为字符串参数接受并返回一个 pattern 对象。注意这里没去匹配字符串。编译正则表达式和以特定字符串 (`'M'`) 进行匹配不是一回事，所牵扯的只是正则表达式本身。
[3]	`re.compile` 返回的已编译的 pattern 对象有几个值得关注的功能：包括了几个 `re` 模块直接提供的功能 (比如： `search` 和 `sub`)。
[4]	用 `'M'` 作参数来调用已编译的 pattern 对象的 `search` 函数与用正则表达式和字符串 `'M'` 调用 `re.search` 可以得到相同的结果，只是快了很多。 (事实上， `re.search` 函数仅仅将正则表达式编译，然后为你调用编译后的 pattern 对象的 `search` 方法。)

> 注意
>
> 在需要多次使用同一个正则表达式的情况下，应该将它进行编译以获得一个 pattern 对象，然后直接调用这个 pattern 对象的方法。

例 15.11. `roman81.py` 中已编译的正则表达式

这个文件可以在例子目录下的 `py/roman/stage8/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
# toRoman and rest of module omitted for clarity
romanNumeralPattern = \
    re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$')
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not romanNumeralPattern.search(s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

[1]	看起来很相似，但实质却有很大改变。 `romanNumeralPattern` 不再是一个字符串了，而是一个由 `re.compile` 返回的 pattern 对象。
[2]	这意味着你可以直接调用 `romanNumeralPattern` 的方法。这比每次调用 `re.search` 要快很多。模块被首次导入 (import) 之时，正则表达式被一次编译并存储于 `romanNumeralPattern` 。之后每次调用 `fromRoman` 时，你可以立刻以正则表达式匹配输入的字符串，而不需要在重复背后的这些编译的工作。

那么编译正则表达式可以提速多少呢？你自己来看吧：

例 15.12. 用 `romantest81.py` 测试 `roman81.py` 的结果

```
.............  ----------------------------------------------------------------------
Ran 13 tests in 3.385s  OK
```

[1]	有一点说明一下：这里，我在运行单元测试时没有 使用 `-v` 选项，因此输出的也不再是每个测试完整的 `doc string` ，而是用一个圆点来表示每个通过的测试。(失败的测试标用 `F` 表示，发生错误则用 `E` 表示，你仍旧可以获得失败和错误的完整追踪信息以便查找问题所在。)
[2]	运行 `13` 个测试耗时 `3.385` 秒，与之相比是没有预编译正则表达式时的 `3.685秒` 。这是一个 `8%` 的整体提速，记住单元测试的大量时间实际上花在做其他工作上。(我单独测试了正则表达式部分的耗时，不考虑单元测试的其他环节，正则表达式编译可以让匹配 `search` 平均提速 `54%` 。)小小修改还真是值得。
[3]	对了，不必顾虑什么，预先编译正则表达式并没有破坏什么，你刚刚证实这一点。

我还想做另外一个性能优化工作。就正则表达式语法的复杂性而言，通常有不止一种方法来构造相同的表达式是不会令人惊讶的。在 comp.lang.python 上对该模块进行一些讨论后，有人建议我使用 `{m,n}` 语法来查找可选重复字符。

例 15.13. `roman82.py`

这个文件可以在例子目录下的 `py/roman/stage8/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
# rest of program omitted for clarity
#old version
#romanNumeralPattern = \
#   re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$')
#new version
romanNumeralPattern = \
    re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')
```

[1]	你已经将 `M?M?M?M?` 替换为 `M{0,4}`。它们的含义相同：“匹配 0 到 4 个 `M` 字符”。类似地，`C?C?C?` 改成了 `C{0,3}` (“匹配 0 到 3 个 `C` 字符”) 接下来的 `X` 和 `I` 也一样。

这样的正则表达简短一些 (虽然可读性不太好)。核心问题是，是否能加快速度？

例 15.14. 以 `romantest82.py` 测试 `roman82.py` 的结果

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s   OK
```

[1]	总体而言，这种正则表达使单元测试提速 2%。这不太令人振奋，但记住 `search` 函数只是整体单元测试的一个小部分，很多时间花在了其他方面。(我另外的测试表明这个应用了新语法的正则表达式使 `search` 函数提速 `11%` 。) 通过预先编译和使用新语法重写可以使正则表达式的性能提升超过 `60%` ，令单元测试的整体性能提升超过 `10%` 。
[2]	比任何的性能提升更重要的是模块仍然运转完好。这便是我早先提到的自由：自由地调整、修改或者重写任何部分并且保证在此过程中没有把事情搞得一团糟。这并不是给无休止地为了调整代码而调整代码以许可；你有很切实的目标 (“让 `fromRoman` 更快”)，而且你可以实现这个目标，不会因为考虑在改动过程中是否会引入新的 Bug 而有所迟疑。

还有另外一个我想做的调整，我保证这是最后一个，之后我会停下来，让这个模块歇歇。就像你多次看到的，正则表达式越晦涩难懂越快，我可不想在六个月内再回头试图维护它。是呀！测试用例通过了，我便知道它工作正常，但如果我搞不懂它是如何 工作的，添加新功能、修正新 Bug，或者维护它都将变得很困难。正如你在 第 7.5 节 “松散正则表达式” 看到的，Python 提供了逐行注释你的逻辑的方法。

例 15.15. `roman83.py`

该文件可以在例子目录下的 `py/roman/stage8/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
# rest of program omitted for clarity
#old version
#romanNumeralPattern = \
#   re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')
#new version
romanNumeralPattern = re.compile('''
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    ''', re.VERBOSE)
```

[1]	`re.compile` 函数的第二个参数是可选的，这个参数通过一个或一组标志 (flag) 来控制预编译正则表达式的选项。这里你指定了 `re.VERBOSE` 选项，告诉 Python 正则表达式里有内联注释。注释和它们周围的空白不 会被认做正则表达式的一部分，在编译正则表达式时 `re.compile` 函数会忽略它们。这个新 “verbose” 版本与老版本完全一样，只是更具可读性。

例 15.16. 用 `romantest83.py` 测试 `roman83.py` 的结果

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s  OK
```

[1]	新 “verbose” 版本和老版本的运行速度一样。事实上，编译的 pattern 对象也一样，因为 `re.compile` 函数会剔除掉所有你添加的内容。
[2]	新 “verbose” 版本可以通过所有老版本通过的测试。什么都没有改变，但在六个月后重读该模块的程序员却有了理解功能如何实现的机会。

15.4. 后记

聪明的读者在学习前一节时想得会更深入一层。现在写的这个程序中最令人头痛的性能负担是正则表达式，但它是必需的，因为没有其它方法来识别罗马数字。但是，它们只有 5000 个，为什么不一次性地构建一个查询表来读取？不必用正则表达式凸现了这个主意的好处。你建立了整数到罗马数字查询表的时候，罗马数字到整数的逆向查询表也构建了。

更大的好处在于，你已经拥有一整套完全的单元测试。你修改了多半的代码，但单元测试还是一样的，因此你可以确定你的新代码与来的代码一样可以正常工作。

例 15.17. `roman9.py`

这个文件可以在例子目录下的 `py/roman/stage9/` 目录中找到。

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
#Roman numerals must be less than 5000
MAX_ROMAN_NUMERAL = 4999
#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))
#Create tables for fast conversion of roman numerals.
#See fillLookupTables() below.
toRomanTable = [ None ]  # Skip an index since Roman numerals have no zero
fromRomanTable = {}
def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n <= MAX_ROMAN_NUMERAL):
        raise OutOfRangeError, "number out of range (must be 1..%s)" % MAX_ROMAN_NUMERAL
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"
    return toRomanTable[n]
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, "Input can not be blank"
    if not fromRomanTable.has_key(s):
        raise InvalidRomanNumeralError, "Invalid Roman numeral: %s" % s
    return fromRomanTable[s]
def toRomanDynamic(n):
    """convert integer to Roman numeral using dynamic programming"""
    result = ""
    for numeral, integer in romanNumeralMap:
        if n >= integer:
            result = numeral
            n -= integer
            break
    if n > 0:
        result += toRomanTable[n]
    return result
def fillLookupTables():
    """compute all the possible roman numerals"""
    #Save the values in two global tables to convert to and from integers.
    for integer in range(1, MAX_ROMAN_NUMERAL + 1):
        romanNumber = toRomanDynamic(integer)
        toRomanTable.append(romanNumber)
        fromRomanTable[romanNumber] = integer
fillLookupTables()
```

这样有多快呢？

例 15.18. 用 `romantest9.py` 测试 `roman9.py` 的结果

```
 .............
----------------------------------------------------------------------
Ran 13 tests in 0.791s
OK
```

还记得吗？你原有版本的最快速度是 13 个测试耗时 3.315 秒。当然，这样的比较不完全公平，因为这个新版本需要更长的时间来导入 (当它填充查询表时)。但是导入只需一次，在运行过程中可以忽略。

这个重构的故事的寓意是什么？

- 简洁是美德。
- 特别是使用正则表达式时。
- 并且单元测试给了你大规模重构的信心……即使原有的代码不是你写的。

15.5. 小结

单元测试是一个强大的概念，使用得当的话既可以减少维护成本又可以增加长期项目的灵活性。同样重要的是要意识到单元测试并不是“灵丹妙药”，也不是“银弹”。编写好的测试用例很困难，保持其更新更需要磨练 (特别是当顾客对修复严重的 Bug 大呼小叫之时)。单元测试不是其它形式测试的替代品，比如说功能性测试、集成测试以及可用性测试。但它切实可行且功效明显，一旦相识，你会反问为什么以往没有应用它。

这一章涵盖了很多内容，有很多都不是 Python 所特有的。很多语言都有单元测试框架，都要求你理解相同的基本概念：

- 测试用例的设计方针是目的单一、可以自动运行、互不干扰。
- 在被测试代码编写之前 编写测试用例。
- 编写测试有效输入的测试用例并检查正确的结果。
- 编写测试无效输入的测试用例并检查正确的失败。
- 为描述 Bug 或反映新需求而编写和升级测试用例。
- 为改进性能、可伸缩性、可读性、可维护性和任何缺少的特性而无情地重构。

另外，你应该能够自如地做到如下 Python 的特有工作：

- 继承 `unittest.TestCase` 生成子类并为每个单独的测试用例编写方法。
- 使用 `assertEqual` 检查已知结果的返回。
- 使用 `assertRaises` 检查函数是否引发已知异常。
- 在 `if __name__` 子句中调用 `unittest.main()` 来一次性运行所有测试用例。
- 以详细 (verbose) 或者普通 (regular) 模式运行单元测试

进一步阅读

- XProgramming.com 有多种语言的 单元测试框架 的下载链接。

第 16 章 函数编程

16.1. 概览

在 第 13 章 单元测试 中，你学会了单元测试的哲学。在 第 14 章 测试优先编程 中你步入了 Python 基本的单元测试操作，在 第 15 章 重构 部分，你看到单元测试如何令大规模重构变得容易。本章将在这些程序样例的基础上，集中关注于超越单元测试本身的更高级的 Python 特有技术。

下面是一个作为简单回归测试 (regression test) 框架运行的完整 Python 程序。它将你前面编写的单独单元测试模块组织在一起成为一个测试套件并一次性运行。实际上这是本书的构建代码的一部分；我为几个样例程序都编写了单元测试 (不是只有 第 13 章 单元测试 中的 `roman.py` 模块)，我的自动构建代码的第一个工作便是确保我所有的例子可以正常工作。如果回归测试程序失败，构建过程当即终止。我可不想因为发布了不能工作的样例程序而让你在下载他们后坐在显示器前抓耳挠腮地为程序不能运转而烦恼。

例 16.1. `regression.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
"""Regression testing framework
This module will search for scripts in the same directory named
XYZtest.py.  Each such script should be a test suite that tests a
module through PyUnit.  (As of Python 2.1, PyUnit is included in
the standard library as "unittest".)  This script will aggregate all
found test suites into one big test suite and run them all at once.
"""
import sys, os, re, unittest
def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
    load = unittest.defaultTestLoader.loadTestsFromModule
    return unittest.TestSuite(map(load, modules))
if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest")
```

把这段代码放在本书其他样例代码相同的目录下运行之，`_` module `_test.py` 中的所有单元测试将被找到并一起被运行。

例 16.2. `regression.py` 的样例输出

```
[you@localhost py]$ python regression.py -v
help should fail with no object ... ok  help should return known result for apihelper ...
help should honor collapse argument ... ok
help should honor spacing argument ... ok
buildConnectionString should fail with list input ... ok  buildConnectionString should fa
buildConnectionString should fail with tuple input ... ok
buildConnectionString handles empty dictionary ... ok
buildConnectionString returns known result with known input ... ok
fromRoman should only accept uppercase input ... ok  toRoman should always return upperca
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
kgp a ref test ... ok
kgp b ref test ... ok
kgp c ref test ... ok
kgp d ref test ... ok
kgp e ref test ... ok
kgp f ref test ... ok
kgp g ref test ... ok
----------------------------------------------------------------------
Ran 29 tests in 2.799s
OK
```

[1]	前五个测试来自于 `apihelpertest.py`，用以测试 第 4 章 自省的威力 中的样例代码。
[2]	接下来的五个测试来自于 `odbchelpertest.py`，用以测试 第 2 章 第一个 *Python* 程序 中的样例代码。
[3]	其他的测试来自于 `romantest.py`，你在 第 13 章 单元测试 中深入学习过。

16.2. 找到路径

从命令行运行 Python 代码时，知道所运行代码在磁盘上的存储位置有时候是有必要的。

这是一个不那么容易想起，但一想起就很容易解决的小麻烦。答案是 `sys.argv`。正如你在 第 9 章 *XML* 处理 中看到的，它包含了很多命令行参数。它也同样记录了运行脚本的名字，和你调用它时使用的命令一摸一样。这些信息足以令我们确定文件的位置。

例 16.3. `fullpath.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
 import sys, os
print 'sys.argv[0] =', sys.argv[0]
pathname = os.path.dirname(sys.argv[0])         print 'path =', pathname
print 'full path =', os.path.abspath(pathname)
```

[1]	无论如何运行一段脚本， `sys.argv[0]` 总是包含脚本的名字，和调用时使用的命令一摸一样。你很快会发现，它不一定包含任何路径信息。
[2]	`os.path.dirname` 接受作为字符串传来的文件名并返回路径部分。如果给定的文件名不包含任何路径信息， `os.path.dirname` 返回空字符串。
[3]	`os.path.abspath` 是这里的关键。它接受的路径名可以是部分的甚至是完全空白，但总能返回完整有效的路径名。

进一步地解释 `os.path.abspath` 是有必要的。它非常灵活，可以接受任何类型的路径名。

例 16.4. `os.path.abspath` 的进一步解释

```
>>> import os
>>> os.getcwd()
/home/you
>>> os.path.abspath('')
/home/you
>>> os.path.abspath('.ssh')
/home/you/.ssh
>>> os.path.abspath('/home/you/.ssh')
/home/you/.ssh
>>> os.path.abspath('.ssh/../foo/')
/home/you/foo
```

[1]	`os.getcwd()` 返回当前的工作路径。
[2]	用空字符串调用 `os.path.abspath` 将返回当前的工作路径，与 `os.getcwd()` 的效果相同。
[3]	以不完整的路径名调用 `os.path.abspath` 可以构建一个基于当前工作路径且完整有效的路径名。
[4]	以完整的路径名调用 `os.path.abspath` 则简单地将其直接返回。
[5]	`os.path.abspath` 还格式化 返回的路径名。注意这个例子在我根本没有‘foo’目录时同样奏效。 `os.path.abspath` 从不检查你的磁盘，而仅仅是字符串操作。

> 注意
>
> 传递给 `os.path.abspath` 的路径名和文件名可以不存在。
>
> 注意
>
> `os.path.abspath` 不仅构建完整路径名，还能格式化路径名。这意味着如果你正工作于 `/usr/` 目录， `os.path.abspath('bin/../local/bin')` 将会返回 `/usr/local/bin` 。它把路径名格式化为尽可能简单的形式。如果你只是希望简单地返回这样的格式化路径名而不需要完整路径名，可以使用 `os.path.normpath` 。

例 16.5. `fullpath.py` 的样例输出

```
[you@localhost py]$ python /home/you/diveintopython/common/py/fullpath.py
sys.argv[0] = /home/you/diveintopython/common/py/fullpath.py
path = /home/you/diveintopython/common/py
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ python common/py/fullpath.py
sys.argv[0] = common/py/fullpath.py
path = common/py
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ cd common/py
[you@localhost py]$ python fullpath.py
sys.argv[0] = fullpath.py
path =
full path = /home/you/diveintopython/common/py
```

[1]	在第一种情况下， `sys.argv[0]` 包含代码的完整路径。你可以通过 `os.path.dirname` 函数将文件名从其中剥离出来并返回完整的路径， `os.path.abspath` 则是简单地把你传递给它的值返回。
[2]	如果脚本是以不完整路名被运行的， `sys.argv[0]` 还是会包含命令行中出现的一切。 `os.path.dirname` 将会给你一个 (相对于当前工作路径的) 不完整的路径名， `os.path.abspath` 将会以不完整路径名为基础构建一个完整的路径名。
[3]	如果没有给定任何路径，而是从当前目录运行脚本， `os.path.dirname` 将简单地返回一个空字符串。由于是从当前目录运行脚本， `os.path.abspath` 将针对给定的空字符串给出你所希望获知的当前目录。

> 注意
> 就像 `os` 和 `os.path` 模块的其他函数，`os.path.abspath` 是跨平台的。如果你是在 Windows (使用反斜杠作为路径符号) 或 Mac OS (使用冒号) 上运行，它们同样工作，只是将获得与我稍有不同的结果。`os` 的所有函数都是这样的。

补充. 一位读者对这个结果并不满意，他希望能够从当前路径运行所有单元测试，而不是从 `regression.py` 所在目录运行。他建议以下面的代码加以取代：

例 16.6. 在当前目录运行脚本

```
import sys, os, re, unittest
def regressionTest():
    path = os.getcwd()
    sys.path.append(path)
    files = os.listdir(path)
```

[1]	不是将 `path` 设置为运行代码所在的路径，而是将它设置为当前目录。可以是你在运行脚本之前所在的任何路径，而不需要是运行脚本所在的路径。(多次体味这句话，直到你真正理解了它。)
[2]	将这个目录添加到 Python 库搜索路径中，你稍后动态导入单元测试模块时，Python 就能找到它们了。如果 `path` 就是正在运行代码的存储目录，你就不需要这样做了，因为 Python 总会查找这个目录。
[3]	函数的其他部分不变。

这个技术允许你在多个项目中重用 `regression.py` 代码。只需要将这个代码放在一个普通目录中，在运行项目前将路径更改为项目的目录。项目中所有的单元测试被找到并运行，而不仅仅局限于 `regression.py` 所在目录的单元测试。

16.3. 重识列表过滤

你已经熟识了应用列表解析来过滤列表。这里介绍的是达到相同效果的另一种令很多人感觉清晰的实现方法。

Python 有一个内建 `filter` 函数，它接受两个参数：一个函数和一个列表，返回一个列表。[12] 作为第一个参数传递给 `filter` 的函数本身应接受一个参数，`filter` 返回的列表将会包含被传入列表参数传递给 `filter` 所有可以令函数返回真 (true) 的元素。

都明白了吗？并没有听起来那么难。

例 16.7. `filter` 介绍

```
>>> def odd(n):
... return n % 2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> filter(odd, li)
[1, 3, 5, 9, -3]
>>> [e for e in li if odd(e)]
>>> filteredList = []
>>> for n in li:
... if odd(n):
...         filteredList.append(n)
...
>>> filteredList
[1, 3, 5, 9, -3]
```

[1]	`odd` 使用内建的取模 (mod) 函数 “ `%` ” 对于为奇数的 `n` 返回 `1` ；为偶数的返回 `0` 。
[2]	`filter` 接受两个参数：一个函数 (`odd`) 和一个列表 (`li`)。它依列表循环为每个元素调用 `odd` 函数。如果 `odd` 返回的是真 (记住，Python 认为所有非零值为真)，则该元素被放在返回列表中，如若不然则被过滤掉。结果是一个只包含原列表中奇数的列表，出现顺序则和原列表相同。
[3]	你可以通过遍历的方式完成相同的工作，正如在 第 4.5 节 “过滤列表” 中看到的。
[4]	你可以通过 `for` 循环的方式完成相同的工作。取决于你的编程背景，这样也许更“直接”，但是像 `filter` 函数这样的实现方法更清晰。不但编写简单，而且易于读懂。 `for` 循环就好比近距离的绘画：你可以看到所有的细节，但是或许你应该花几秒时间退后几步看一看图画的全景：“啊，你仅仅是要过滤列表！”

例 16.8. `regression.py` 中的 `filter`

```
files = os.listdir(path)
test = re.compile("test\.py$", re.IGNORECASE)
files = filter(test.search, files)
```

[1]	正如你在 第 16.2 节 “找到路径” 中看到的, `path` 可能包括正在运行脚本的完全或者部分路径名，或者当脚本运行自当前目录时包含一个空的字符串。任何一种情况下， `files` 都会获得正运行脚本所在目录的文件名。
[2]	这是一个预编译的正则表达式。正如你在 第 15.3 节 “重构”中看到的，如果你需要反复使用同一个正则表达式，你应该编译它已获得更快的性能。编译后的对象将含有接受一个待寻找字符串作为参数的 `search` 方法。如果这个正则表达式匹配字符串， `search` 方法返回一个包含正则表达式匹配信息的 `Match` 对象；否则返回 `None` ，这是 Python 空 (null) 值。
[3]	对于 `files` 列表中的每个元素，你将会调用正则表达式编译对象 `test` 的 `search` 方法。如果正则表达匹配，方法将会返回一个被 Python 认定为真 (true) 的 `Match` 对象；如果正则表达不匹配， `search` 方法将会返回被认定为假 (false) 的 `None` ，元素将被排除。

历史注释. Python 2.0 早期的版本不包含 列表解析，因此不能 以列表解析方式过滤， `filter` 函数是当时唯一的方法。即便是在引入列表解析的 2.0 版，有些人仍然钟情于老派的 `filter` (和这章稍后将见到的它的伴侣函数 `map`)。两种方法并存于世，使用哪种方法只是风格问题， `map` 和 `filter` 将在未来的 Python 版本中被废止的讨论尚无定论。

例 16.9. 以列表解析法过滤

```
files = os.listdir(path)
test = re.compile("test\.py$", re.IGNORECASE)
files = [f for f in files if test.search(f)]
```

[1]	这种方法将完成和 `filter` 函数完全相同的工作。哪种方法更清晰完全取决于你自己。

Footnotes

[12] 从技术层面上讲， `filter` 的第二个参数可以是任意的序列，包括列表、元组以及定义了 `__getitem__` 特殊方法而能像列表一样工作的自定义类。在可能情况下， `filter` 会返回与输入相同的数据类型，也就是过滤一个列表返回一个列表，过滤一个元组返回一个元组。

16.4. 重识列表映射

你对使用列表解析映射列表的做法已经熟知。另一种方法可以完成同样的工作：使用内建 `map` 函数。它的工作机理和 `filter` 函数类似。

例 16.10. `map` 介绍

```
>>> def double(n):
... return n*2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> map(double, li)
[2, 4, 6, 10, 18, 20, 512, -6]
>>> [double(n) for n in li]
[2, 4, 6, 10, 18, 20, 512, -6]
>>> newlist = []
>>> for n in li:
... newlist.append(double(n))
...
>>> newlist
[2, 4, 6, 10, 18, 20, 512, -6]
```

[1]	`map` 接受一个函数和一个列表作为参数，[13] 并对列表中的每个元素依次调用函数返回一个新的列表。在这个例子中，函数仅仅是将每个元素乘以 2。
[2]	使用列表解析的方法你可以做到相同的事情。列表解析是在 Python 2.0 版时被引入的；而 `map` 则古老得多。
[3]	你如果坚持以 Visual Basic 程序员自居，通过 `for` 循环的方法完成相同的任务也完全可以。

例 16.11. `map` 与混合数据类型的列表

```
>>> li = [5, 'a', (2, 'b')]
>>> map(double, li)
[10, 'aa', (2, 'b', 2, 'b')]
```

[1]	作为一个旁注，我想指出只要提供的那个函数能够正确处理各种数据类型，`map` 对于混合数据类型列表的处理同样出色。在这里，`double` 函数仅仅是将给定参数乘以 2，Python 则会根据参数的数据类型决定正确操作的方法。对整数而言，这意味着乘 2；对字符串而言，意味着把自身和自身连接；对于元组，意味着构建一个包括原始元组全部元素和原始元组组合在一起的新元组。

好了，玩够了。让我们来看一些真实代码。

例 16.12. `regression.py` 中的 `map`

```
filenameToModuleName = lambda f: os.path.splitext(f)[0]
moduleNames = map(filenameToModuleName, files)
```

[1]	正如你在 第 4.7 节 “使用 lambda 函数” 中所见， `lambda` 定义一个内联函数。也正如你在 例 6.17 “分割路径名” 中所见， `os.path.splitext` 接受一个文件名并返回一个元组 `(_name_, _extension_)` 。因此 `filenameToModuleName` 是一个接受文件名，剥离出其扩展名，然后只返回文件名称的函数。
[2]	调用 `map` 将把 `files` 列出的所有文件名传递给 `filenameToModuleName` 函数，并且返回每个函数调用结果所组成的列表。换句话说，你剔除掉文件名的扩展名，并将剔除后的文件名存于 `moduleNames` 之中。

如你在本章剩余部分将看到的，你可以将这种数据中心思想扩展到定义和执行一个容纳来自很多单个测试套件的测试的一个测试套件的最终目标。

Footnotes

[13] 同前，我需要指出 `map` 可以接受一个列表、元组，或者一个像序列一样的对象。参见前面的关于 `filter` 的脚注。

16.5. 数据中心思想编程

现在的你，可能正抓耳挠腮地狠想，为什么这样比使用 `for` 循环和直接调用函数好。这是一个非常好的问题。通常这是一个程序观问题。使用 `map` 和 `filter` 强迫你围绕数据进行思考。

就此而言，你从没有数据开始，你所做的第一件事是获得当前脚本的目录路径，并获得该目录中的文件列表。这就是关键的一步，使你有了待处理的真实数据：文件名列表。

当然，你知道你并不关心所有的文件，而只关心测试套件。你有太多数据，因此你需要过滤(`filter`)数据。你如何知道哪些数据应该保留？你需要一个测试来确定，因此你定义一个测试并把它传给 `filter` 函数。这里你应用了一个正则表达式来确定，但无论如何构建测试，原则是一样的。

现在你有了每个测试套件的文件名 (且局限于测试套件，因为所有其他内容都被过滤掉了)，但是你还需要以模块名来替代之。你有正确数量的数据，只是格式不正确。因此，你定义了一个函数来将文件名转换为模块名，并使用这个函数映射整个列表。从一个文件名，你可以获得一个模块名，从一个文件名列表，你可以获得一个模块名列表。

如果不应用 `filter`，你也可以使用 `for` 循环结合一个 `if` 语句的方法。 `map` 的使用则可以由一个 `for` 循环和一个函数调用来取代。但是 `for` 循环看起来像是个繁重的工作。至少，简单讲是在浪费时间，糟糕的话还会隐埋 Bug。例如，你需要弄清楚如何测试这样一个条件：“这个文件是测试套件吗？”这是应用特定的逻辑，没有哪个语言能自动为我们写出其代码。但是一旦你搞清楚了，你还需要费尽周折地定义一个新的空列表，写一个 `for` 循环以及一个 `if` 语句并手工地调用 `append` 将符合条件的元素一个个添加到新列表中，然后一路上注意区分哪个变量里放着过滤后的数据，哪个变量里放着未过滤的老数据。为什么不直接定义测试条件，然后由 Python 为你完成接下来的工作呢？

当然啦，你可以尝试眩一点的做法，去删除列表中的元素而不新建一个列表。但是你以前吃过这样的亏。试图在循环中改变数据结构是很容易出问题的。Python 是一个这样工作的语言吗？用多长时间你才能搞清这一点？你能确定记得你第二次这样尝试的安全性？程序员在和这类纯技术课题较劲的过程中，花费了太多的时间，犯了太多的错误，却并没有什么意义。这样并不可能令你的程序有所进步，只不过是费力不讨好。

我在第一次学习 Python 时是抵触列表解析的，而且我抗拒 `filter` 和 `map` 的时间更长。我坚持着我更艰难的生活，固守着类似于 `for` 循环和 `if` 语句以及一步步地以代码为中心的编程方式。而且我的 Python 程序看起来很像是 Visual Basic 程序，细化每一个函数中的每一个操作步骤。它们却有着同样的小错误和隐蔽的 Bug。这一切其实都没有意义。

让这一切都远去吧。费力不讨好的编程不重要，数据重要。并且数据并不麻烦，它们不过就是数据。如果多了，就过滤。如果不是我们要的，就映射。聚焦在数据上，摒弃费力的劳作。

16.6. 动态导入模块

好了，大道理谈够了。让我们谈谈动态导入模块吧。

首先，让我们看一看正常的模块导入。 `import _module_` 语法查看搜索路径，根据给定的名字寻找模块并导入它们。你甚至可以这样做：以逗号分割同时导入多个模块，本章代码前几行就是这样做的。

例 16.13. 同时导入多个模块

```
import sys, os, re, unittest
```

[1]	这里同时导入四个模块： `sys` (为系统函数和得到命令行参数)、 `os` (为目录列表之类的操作系统函数)、 `re` (为正则表达式)，以及 `unittest` (为单元测试)。

现在让我们用动态导入做同样的事。

例 16.14. 动态导入模块

```
>>> sys = __import__('sys')
>>> os = __import__('os')
>>> re = __import__('re')
>>> unittest = __import__('unittest')
>>> sys
>>> <module 'sys' (built-in)>
>>> os
>>> <module 'os' from '/usr/local/lib/python2.2/os.pyc'>
```

[1]	内建 `__import__` 函数与 `import` 语句的既定目标相同，但它是一个真正的函数，并接受一个字符串参数。
[2]	变量 `sys` 现在是 `sys` 模块，和 `import sys` 的结果完全相同。变量 `os` 现在是一个 `os` 模块，等等。

因此 `__import__` 导入一个模块，但是是通过一个字符串参数来做到的。依此处讲，你用以导入的仅仅是一个硬编码性的字符串，但它可以是一个变量，或者一个函数调用的结果。并且你指向模块的变量也不必与模块名匹配。你可以导入一系列模块并把它们指派给一个列表。

例 16.15. 动态导入模块列表

```
>>> moduleNames = ['sys', 'os', 're', 'unittest']
>>> moduleNames
['sys', 'os', 're', 'unittest']
>>> modules = map(__import__, moduleNames)
>>> modules
[<module 'sys' (built-in)>,
<module 'os' from 'c:\Python22\lib\os.pyc'>,
<module 're' from 'c:\Python22\lib\re.pyc'>,
<module 'unittest' from 'c:\Python22\lib\unittest.pyc'>]
>>> modules[0].version
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
>>> import sys
>>> sys.version
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
```

[1]	`moduleNames` 只是一个字符串列表。没什么特别的，只是这些名字刚好是你可应需而用的可导入模块名。
[2]	简单得令人惊奇，通过映射 `__import__` 就实现了导入。记住，列表 (`moduleNames`) 的每个元素将被用来一次次调用函数 (`__import__`) 并以一个返回值构成的列表作为返回结果。
[3]	所以现在你已经由一个字符串列表构建起了一个实际模块的列表。(你的路径可能不同，这取决于你的操作系统、你安装 Python 的位置、月亮残缺的程度等等。)
[4]	从这些是真实模块这一点出发，让我们来看一些模块属性。记住， `modules[0]` 是 `sys` 模块，因此， `modules[0].version` 是 `sys.version` 。所有模块的其他属性和方法也都可用。 `import` 语句没什么神奇的，模块也没什么神奇的。模块就是对象，一切都是对象。

现在，你应该能够把这一切放在一起，并搞清楚本章大部分样例代码是做什么的。

16.7. 全部放在一起

你已经学习了足够的知识，现在来分析本章样例代码的前七行：读取一个目录并从中导入选定的模块。

例 16.16. `regressionTest` 函数

```
def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
load = unittest.defaultTestLoader.loadTestsFromModule
return unittest.TestSuite(map(load, modules))
```

让我们一行行交互地看。假定当前目录是 `c:\diveintopython\py`，其中有包含本章脚本在内的本书众多样例。正如在 第 16.2 节 “找到路径” 中所见，脚本目录将存于 `path` 变量，因此让我们从这里开始以实打实的代码起步。

例 16.17. 步骤 1：获得所有文件

```
>>> import sys, os, re, unittest
>>> path = r'c:\diveintopython\py'
>>> files = os.listdir(path)
>>> files
['BaseHTMLProcessor.py', 'LICENSE.txt', 'apihelper.py', 'apihelpertest.py',
'argecho.py', 'autosize.py', 'builddialectexamples.py', 'dialect.py',
'fileinfo.py', 'fullpath.py', 'kgptest.py', 'makerealworddoc.py',
'odbchelper.py', 'odbchelpertest.py', 'parsephone.py', 'piglatin.py',
'plural.py', 'pluraltest.py', 'pyfontify.py', 'regression.py', 'roman.py', 'romantest.py'
'uncurly.py', 'unicode2koi8r.py', 'urllister.py', 'kgp', 'plural', 'roman',
'colorize.py']
```

[1]	`files` 是由脚本所在目录的所有文件和目录构成的列表。(如果你已经运行了其中的一些样例，可能还会看到一些 `.pyc` 文件。)

例 16.18. 步骤 2：找到你关注的多个文件

```
>>> test = re.compile("test\.py$", re.IGNORECASE)
>>> files = filter(test.search, files)
>>> files
['apihelpertest.py', 'kgptest.py', 'odbchelpertest.py', 'pluraltest.py', 'romantest.py']
```

[1]	这个正则表达式将匹配以 `test.py` 结尾的任意字符串。注意，你必须转义这个点号，因为正则表达式中的点号通常意味着 “匹配任意单字符”，但是你实际上想匹配的事一个真正的点号。
[2]	被编译的正则表达式就像一个函数，因此你可以用它来过滤文件和目录构成的大列表，找寻符合正则表达式的所有元素。
[3]	剩下的是一个单元测试脚本列表，因为只有它们是形如 `SOMETHINGtest.py` 的文件。

例 16.19. 步骤 3：映射文件名到模块名

```
>>> filenameToModuleName = lambda f: os.path.splitext(f)[0]
>>> filenameToModuleName('romantest.py')
'romantest'
>>> filenameToModuleName('odchelpertest.py')
'odbchelpertest'
>>> moduleNames = map(filenameToModuleName, files)
>>> moduleNames
['apihelpertest', 'kgptest', 'odbchelpertest', 'pluraltest', 'romantest']
```

[1]	正如你在 第 4.7 节 “使用 lambda 函数” 中所见， `lambda` 快餐式地创建内联单行函数。这里应用你在 例 6.17 “分割路径名” 中已经见过的，标准库的 `os.path.splitext` 将一个带有扩展名的文件名返回成只包含文件名称的那部分。
[2]	`filenameToModuleName` 是一个函数。 `lambda` 函数并不比你以 `def` 语句定义的普通函数神奇。你可以如其他函数一样地调用 `filenameToModuleName` ，它也将如你所愿：从参数中剔除扩展名。
[3]	现在你可以通过 `map` 把这个函数应用于单元测试文件列表中的每一个文件。
[4]	结果当然如你所愿：以指代模块的字符串构成的一个列表。

例 16.20. 步骤 4：映射模块名到模块

```
>>> modules = map(__import__, moduleNames)
>>> modules
[<module 'apihelpertest' from 'apihelpertest.py'>,
<module 'kgptest' from 'kgptest.py'>,
<module 'odbchelpertest' from 'odbchelpertest.py'>,
<module 'pluraltest' from 'pluraltest.py'>,
<module 'romantest' from 'romantest.py'>]
>>> modules[-1]
<module 'romantest' from 'romantest.py'>
```

[1]	正如你在 第 16.6 节 “动态导入模块” 中所见，你可以通过 `map` 和 `__import__` 的协同工作，将模块名 (字符串) 映射到实际的模块 (像其他模块一样可以被调用和使用)。
[2]	`modules` 现在是一个模块列表，其中的模块和其他模块一样。
[3]	该列表的最后一个模块是 `romantest` 模块，和通过 `import romantest` 导入的模块完全等价。

例 16.21. 步骤 5：将模块载入测试套件

```
>>> load = unittest.defaultTestLoader.loadTestsFromModule
>>> map(load, modules)
[<unittest.TestSuite tests=[
  <unittest.TestSuite tests=[<apihelpertest.BadInput testMethod=testNoObject>]>,
  <unittest.TestSuite tests=[<apihelpertest.KnownValues testMethod=testApiHelper>]>,
  <unittest.TestSuite tests=[
    <apihelpertest.ParamChecks testMethod=testCollapse>,
    <apihelpertest.ParamChecks testMethod=testSpacing>]>,
    ...
  ]
]
>>> unittest.TestSuite(map(load, modules))
```

[1]	模块对象的存在，使你不但可以像其他模块一样地使用它们；通过类的实例化和函数的调用，你还可以内省模块，从而弄清楚已经有了那些类和函数。这正是 `loadTestsFromModule` 方法的工作：内省每一个模块并为每个模块返回一个 `unittest.TestSuite` 对象。每个 `TestSuite` (测试套件) 对象都包含一个 `TestCase` 对象的列表，每个对象对应着你的模块中的一个测试方法。
[2]	最后，你将 `TestSuite` 列表封装成一个更大的测试套件。 `unittest` 模块会很自如地遍历嵌套于测试套件中的树状结构，最后深入到独立测试方法，一个个加以运行并判断通过或是失败。

自省过程是 `unittest` 模块经常为我们做的一项工作。还记得我们的独立测试模块仅仅调用了看似神奇的 `unittest.main()` 函数就大刀阔斧地完成了全部工作吗？`unittest.main()` 实际上创建了一个 `unittest.TestProgram` 的实例，而这个实例实际上创建了一个 `unittest.defaultTestLoader` 的实例并以调用它的模块启动它。 (如果你不给出，如何知道调用它的模块是哪一个？通过使用同样神奇的 `__import__('__main__')` 命令，动态导入正在运行的模块。我可以就 `unittest` 模块中使用的所有技巧和技术写一本书，但那样我就没法写完这本了。)

例 16.22. 步骤 6：告知 `unittest` 使用你的测试套件

```
if __name__ == "__main__":
   unittest.main(defaultTest="regressionTest")
```

[1]	在不使用 `unittest` 模块来为我们做这一切的神奇工作的情况下，你实际上已自己做到了。你已经创建了一个自己就能导入模块、调用 `unittest.defaultTestLoader` 并封装于一个测试套件的 `regressionTest` 函数。现在你所要做的不是去寻找测试并以通用的方法构建一个测试套件，而是告诉 `unittest` 前面那些，它将调用 `regressionTest` 函数，而它会返回可以直接使用的 `TestSuite` 。

16.8. 小结

`regression.py` 程序及其输出到现在应该很清楚了。

你现在应该能够很自如地做到如下事情：

- 从命令行操作路径信息。
- 不使用列表解析，使用 `filter` 过滤列表。
- 不使用列表解析，使用 `map` 映射列表。
- 动态导入模块。

第 17 章 动态函数

17.1. 概览

我想谈谈名词复数。还有，返回其它函数的函数，高级的正则表达式和生成器 (Generator)。生成器是 Python 2.3 新引入的。但首先还是让我们先来谈谈如何生成名词复数。

如果你还没有看过 第 7 章 正则表达式，现在是个绝佳的机会。这章中假定你已理解了正则表达式的基础内容并迅速深入更高级的应用。

英语是一个吸收很多外来语而令人疯掉的语言，把单数名词变成复数的规则则是复杂而又多变的。有规则，有例外，更有例外的例外。

如果你在英语国家长大或是在正规学校学习了英语，你可能对下面的基本规则很熟悉：

1. 如果一个词以 S, X 或 Z 结尾，加 ES。如 “Bass” 变成 “basses”，“fax” 变成 “faxes”，还有 “waltz” 变成 “waltzes”。
2. 如果一个词以发音的 H 结尾，加 ES；若以不发音的 H 结尾，加 S。什么是发音的 H？和其他字母混合在一起发出一个你可以听到的声音。那么，“coach” 变成 “coaches”，“rash” 变成 “rashes”，因为在读出来时，你可以听到 CH 和 SH 的声音。但是，“cheetah” 变成 “cheetahs”，因为 H 不发音。
3. 如果一个词以发 I 音的 Y 结尾，把 Y 变成 IES；如果 Y 与元音搭配在一起发出其他声音则只添加 S。因此，“vacancy” 变成 “vacancies”，但 “day” 变成 “days”。
4. 如果一切规则都不适用，就只添加 S 并祈祷不会错。

(我知道有很多例外情况，比如：“Man” 变成 “men”，“woman” 变成 “women”，但是，“human” 却变成 “humans”。“Mouse” 变成 “mice”，“louse” 变成 “lice”，但是，“house” 却变成 “houses”。“Knife” 变成 “knives”，“wife” 变成 “wives”，但是 “lowlife” 却变成 “lowlifes”。更不要说那些复数根本就不需要变化的词了，比如 “sheep”, “deer” 和 “haiku”。)

其他的语言当然完全不同。

让我们来设计一个复数化名词的模块吧！从英语名词开始，仅考虑上面的四种规则，但是记得你将来需要不断添加规则，更可能最后添加进更多的语言。

17.2. `plural.py`，第 1 阶段

你所针对的单词 (至少在英语中) 是字符串和字符。你还需要规则来找出不同的字符 (字母) 组合，并对它们进行不同的操作。这听起来像是正则表达式的工作。

例 17.1. `plural1.py`

```
 import re
def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeioudgkprt]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

[1]	好啦，这是一个正则表达式，但是它使用了你在 第 7 章 正则表达式 中未曾见过的语法。方括号的意思是 "完全匹配这些字符中的一个"。也就是说， `[sxz]` 意味着 " `s` ， 或者 `x` ， 再或者 `z` "，但只是其中的一个。 `$` 应该不陌生，它意味着匹配字符串的结尾。也就是说，检查 `noun` 是否以 `s` ， `x` ， 或者 `z` 结尾。
[2]	`re.sub` 函数进行以正则表达式为基础的替换工作。让我们更具体地看看它。

例 17.2. `re.sub` 介绍

```
>>> import re
>>> re.search('[abc]', 'Mark')
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.sub('[abc]', 'o', 'Mark')
'Mork'
>>> re.sub('[abc]', 'o', 'rock')
'rook'
>>> re.sub('[abc]', 'o', 'caps')
'oops'
```

[1]	`Mark` 包含 `a` ， `b` ， 或者 `c` 吗？是的，含有 `a` 。
[2]	好的，现在找出 `a` ， `b` ， 或者 `c` 并以 `o` 取代之。 `Mark` 就变成 `Mork` 了。
[3]	同一方法可以将 `rock` 变成 `rook` 。
[4]	你可能认为它可以将 `caps` 变成 `oaps` ， 但事实并非如此。 `re.sub` 替换所有 的匹配项，并不只是第一个匹配项。因此正则表达式将会把 `caps` 变成 `oops` ， 因为 `c` 和 `a` 都被转换为 `o` 了。

例 17.3. 回到 `plural1.py`

```
 import re
def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeioudgkprt]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

[1]	回到 `plural` 函数。你在做什么？你在以 `es` 取代字符串的结尾。换句话说，追加 `es` 到字符串。你可以通过字符串拼合做到相同的事，例如 `noun + 'es'` ，但是我使用正则表达式做这一切，既是为了保持一致，也是为了本章稍后你会明白的其它原因。
[2]	仔细看看，这是另一个新的内容。 `^` 是方括号里面的第一个字符，这有特别的含义：否定。 `[^abc]` 意味着 “ 除 `a` 、 `b` 、 和 `c` 以外的 任意单字符”。所以， `[^aeioudgkprt]` 意味着除 `a` 、 `e` 、 `i` 、 `o` 、 `u` 、 `d` 、 `g` 、 `k` 、 `p` 、 `r` 和 `t` 以外的任意字符。这个字符之后应该跟着一个 `h` ，然后是字符串的结尾。你在寻找的是以发音的 H 结尾的单词。
[3]	这是一个相似的表达：匹配 Y 前面不是 `a` 、 `e` 、 `i` 、 `o` 和 `u` ，并以这个 Y 结尾的单词。你在查找的是以发 I 音的 Y 结尾的单词。

例 17.4. 正则表达式中否定的更多应用

```
>>> import re
>>> re.search('[^aeiou]y$', 'vacancy')
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.search('[^aeiou]y$', 'boy')
>>>
>>> re.search('[^aeiou]y$', 'day')
>>>
>>> re.search('[^aeiou]y$', 'pita')
>>>
```

[1]	`vacancy` 匹配这个正则表达式，因为它以 `cy` 结尾，并且 `c` 不在 `a` 、 `e` 、 `i` 、 `o` 和 `u` 之列。
[2]	`boy` 不能匹配，因为它以 `oy` 结尾，并且你特别指出 `y` 之前的字符不可以是 `o` 。 `day` 不能匹配是因为以 `ay` 结尾。
[3]	`pita` 不匹配是因为不以 `y` 结尾。

例 17.5. 更多的 `re.sub`

```
>>> re.sub('y$', 'ies', 'vacancy')
'vacancies'
>>> re.sub('y$', 'ies', 'agency')
'agencies'
>>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy')
'vacancies'
```

[1]	正则表达式把 `vacancy` 变为 `vacancies`，把 `agency` 变为 `agencies`，这正是你想要的。注意，将 `boy` 变成 `boies` 是可行的，但是永远不会发生，因为 `re.search` 首先确定是否应该应用 `re.sub`。
[2]	顺便提一下，可以将两个正则表达式 (一个确定规则适用与否，一个应用规则) 合并在一起成为一个正则表达式。这便是合并后的样子。它的大部分已经很熟悉：你应用的是在 第 7.6 节 “个案研究：解析电话号码” 学过的记忆组 (remembered group) 记住 `y` 之前的字符。然后再替换字符串，你使用一个新的语法 `\1`，这意味着：“嘿！记得前面的第一个组吗？把它放这儿”。就此而言，记住了 `y` 之前的 `c`，然后你做替换工作，你将 `c` 替换到 `c` 的位置，并将 `ies` 替换到 `y` 的位置。(如果你有不止一个组则可以使用 `\2` 或者 `\3` 等等。)

正则表达式替换非常强大，并且 `\1` 语法使之更加强大。但是将整个操作放在一个正则表达式中仍然晦涩难懂，也不能与前面描述的复数规则直接呼应。你原来列出的规则，比如 “如果单词以 S，X 或者 Z 结尾，结尾追加 ES”。如果你在函数中看到两行代码描述 “如果单词以 S，X 或者 Z 结尾，结尾追加 ES”，更加直观些。

17.3. `plural.py` ，第 2 阶段

现在你将增加一个抽象过程。你从定义一个规则列表开始：如果这样，就做那个，否则判断下一规则。让我们暂时将程序一部分复杂化以便使另一部分简单化。

例 17.6. `plural2.py`

```
 import re
def match_sxz(noun):
    return re.search('[sxz]$', noun)
def apply_sxz(noun):
    return re.sub('$', 'es', noun)
def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)
def apply_h(noun):
    return re.sub('$', 'es', noun)
def match_y(noun):
    return re.search('[^aeiou]y$', noun)
def apply_y(noun):
    return re.sub('y$', 'ies', noun)
def match_default(noun):
    return 1
def apply_default(noun):
    return noun + 's'
rules = ((match_sxz, apply_sxz),
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
         )
def plural(noun):
    for matchesRule, applyRule in rules:
        if matchesRule(noun):
            return applyRule(noun)
```

[1]	这个版本看起来更加复杂 (至少是长了)，但做的工作没有变化：试图顺序匹配四种不同规则，并在匹配时应用恰当的正则表达式。不同之处在于，每个独立的匹配和应用规则都在自己的函数中定义，并且这些函数列于 `rules` 变量这个元组的元组之中。
[2]	使用一个 `for` 循环，你可以根据 `rules` 元组一次性进行匹配和应用规则两项工作 (一个匹配和一个应用)。 `for` 循环第一轮中， `matchesRule` 将使用 `match_sxz` ， `applyRule` 将使用 `apply_sxz` ；在第二轮中 (假设真走到了这么远)， `matchesRule` 将被赋予 `match_h` ， `applyRule` 将被赋予 `apply_h` 。
[3]	记住 Python 中的一切都是对象，包括函数。 `rules` 包含函数；不是指函数名，而是指函数本身。当 `matchesRule` 和 `applyRule` 在 `for` 循环中被赋值后，它们就成了你可以调用的真正函数。因此，在 `for` 循环第一轮中，这就相当于调用 `matches_sxz(noun)` 。
[4]	在 `for` 循环第一轮中，这就相当于调用 `apply_sxz(noun)` ，等等。

这个抽象过程有些令人迷惑，试着剖析函数看看实际的等价内容。这个 `for` 循环相当于：

例 17.7. 剖析 `plural` 函数

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

这里的好处在于 `plural` 函数现在被简化了。它以普通的方法反复使用其它地方定义的规则。获得一个匹配规则，匹配吗？调用并应用规则。规则可以在任意地方以任意方法定义，`plural` 函数对此并不关心。

现在，添加这个抽象过程值得吗？嗯……还不值。让我们看看如何向函数添加一个新的规则。啊哈，在先前的范例中，需要向 `plural` 函数添加一个 `if` 语句；在这个例子中，需要增加两个函数：`match_foo` 和 `apply_foo`，然后更新 `rules` 列表指定在什么相对位置调用这个新匹配和新规则应用。

这其实不过是步入下一节的一个基石。让我们继续。

17.4. `plural.py` , 第 3 阶段

将每个匹配和规则应用分别制作成函数没有必要。你从来不会直接调用它们：你把它们定义于 `rules` 列表之中并从那里调用它们。让我们隐去它们的函数名而抓住规则定义的主线。

例 17.8. `plural3.py`

```
 import re
rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )
def plural(noun):
    for matchesRule, applyRule in rules:
        if matchesRule(noun):
            return applyRule(noun)
```

[1]	这与第 2 阶段定义的规则是一样的。惟一的区别是不再定义 `match_sxz` 和 `apply_sxz` 之类的函数，而是以 lambda 函数 法将这些函数的内容直接 “嵌入” `rules` 列表本身。
[2]	注意 `plural` 函数完全没有变化，还是反复于一系列的规则函数，检查第一个匹配规则，如果返回真则调用第二个应用规则并返回值。和前面一样，给定单词返回单词。唯一的区别是规则函数被内嵌定义，化名作 lambda 函数。但是 `plural` 函数并不在乎它们是如何定义的，只是拿到规则列表，闭着眼睛干活。

现在添加一条新的规则，所有你要做的就是直接在 `rules` 列表之中定义函数：一个匹配规则，一个应用规则。这样内嵌的规则函数定义方法使得没必要的重复很容易被发现。你有四对函数，它们采用相同的模式。匹配函数就是调用 `re.search` ，应用函数就是调用 `re.sub` 。让我们提炼出这些共同点。

17.5. `plural.py` , 第 4 阶段

让我们精炼出代码中的重复之处，以便更容易地定义新规则。

例 17.9. `plural4.py`

```
 import re
def buildMatchAndApplyFunctions((pattern, search, replace)):
    matchFunction = lambda word: re.search(pattern, word)
    applyFunction = lambda word: re.sub(search, replace, word)
    return (matchFunction, applyFunction)
```

[1]	`buildMatchAndApplyFunctions` 是一个动态生成其它函数的函数。它将 `pattern` ， `search` 和 `replace` (实际上是一个元组，我们很快就会提到这一点)，通过使用 `lambda` 语法构建一个接受单参数 (`word`) 并以传递给 `buildMatchAndApplyFunctions` 的 `pattern` 和传递给新函数的 `word` 调用 `re.search` 的匹配函数！哇塞！
[2]	构建应用规则函数的方法相同。应用规则函数是一个接受单参数并以传递给 `buildMatchAndApplyFunctions` 的 `search` 和 `replace` 以及传递给这个应用规则函数的 `word` 调用 `re.sub` 的函数。在一个动态函数中应用外部参数值的技术被称作闭合 *(closures)*。你实际上是在应用规则函数中定义常量：它只接受一个参数 (`word`)，但用到了定义时设置的两个值 (`search` 和 `replace`)。
[3]	最终， `buildMatchAndApplyFunctions` 函数返回一个包含两个值的元组：你刚刚创建的两个函数。你在这些函数中定义的常量 (`matchFunction` 中的 `pattern` 以及 `applyFunction` 中的 `search` 和 `replace`) 保留在这些函数中，由 `buildMatchAndApplyFunctions` 一同返回。这简直太酷了。

如果这太费解 (它应该是这样，这是个怪异的东西)，可能需要通过了解它的使用来搞明白。

例 17.10. `plural4.py` 继续

```
patterns = \
  (
    ('[sxz]$', '$', 'es'),
    ('[^aeioudgkprt]h$', '$', 'es'),
    ('(qu|[^aeiou])y$', 'y$', 'ies'),
    ('$', '$', 's')
  )
rules = map(buildMatchAndApplyFunctions, patterns)
```

[1]	我们的复数化规则现在被定义成一组字符串 (不是函数)。第一个字符串是你在调用 `re.search` 时使用的正则表达式；第二个和第三个字符串是你在通过调用 `re.sub` 来应用规则将名词变为复数时使用的搜索和替换表达式。
[2]	这很神奇。把传进去的 `patterns` 字符串转换为传回来的函数。如何做到的呢？将这些字符串映射给 `buildMatchAndApplyFunctions` 函数之后，三个字符串参数转换成了两个函数组成的元组。这意味着 `rules` 被转换成了前面范例中相同的内容：由许多调用 `re.search` 函数的匹配函数和调用 `re.sub` 的规则应用函数构成的函数组组成的一个元组。

我发誓这不是我信口雌黄：`rules` 被转换成了前面范例中相同的内容。剖析 `rules` 的定义，你看到的是：

例 17.11. 剖析规则定义

```
rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )
```

例 17.12. `plural4.py` 的完成

```
def plural(noun):
    for matchesRule, applyRule in rules:
        if matchesRule(noun):
            return applyRule(noun)
```

[1]	由于 `rules` 列表和前面的范例是相同的，`plural` 函数没有变化也就不令人诧异了。记住，这没什么特别的，按照顺序调用一系列函数。不必在意规则是如何定义的。在第 2 阶段，它们被定义为各具名称的函数。在第 3 阶段，他们被定义为匿名的 `lambda` 函数。现在第 4 阶段，它们通过 `buildMatchAndApplyFunctions` 映射原始的字符串列表被动态创建。无所谓，`plural` 函数的工作方法没有变。

还不够兴奋吧！我必须承认，在定义 `buildMatchAndApplyFunctions` 时我跳过了一个微妙之处。让我们回过头再看一下。

例 17.13. 回头看 `buildMatchAndApplyFunctions`

```
def buildMatchAndApplyFunctions((pattern, search, replace)):
```

[1]	注意到双括号了吗？这个函数并不是真的接受三个参数，实际上只接受一个参数：一个三元素元组。但是在函数被调用时元组被展开了，元组的三个元素也被赋予了不同的变量：`pattern`，`search` 和 `replace`。乱吗？让我们在使用中理解。

例 17.14. 调用函数时展开元组

```
>>> def foo((a, b, c)):
... print c
... print b
... print a
>>> parameters = ('apple', 'bear', 'catnap')
>>> foo(parameters)
catnap
bear
apple
```

[1]	调用 `foo` 的正确方法是使用一个三元素元组。函数被调用时，元素被分别赋予 `foo` 中的多个局部变量。

现在，让我们回过头看一看这个元组自动展开技巧的必要性。`patterns` 是一个元组列表，并且每个元组都有三个元素。调用 `map(buildMatchAndApplyFunctions, patterns)`，这并不 意味着是以三个参数调用 `buildMatchAndApplyFunctions`。使用 `map` 映射一个列表到函数时，通常使用单参数：列表中的每个元素。就 `patterns` 而言，列表的每个元素都是一个元组，所以 `buildMatchAndApplyFunctions` 总是是以元组来调用，在 `buildMatchAndApplyFunctions` 中使用元组自动展开技巧将元素赋值给可以被使用的变量。

17.6. `plural.py`, 第 5 阶段

你已经精炼了所有重复代码，也尽可能地把复数规则提炼到定义一个字符串列表。接下来的步骤是把这些字符串提出来放在另外的文件中，从而可以和使用它们的代码分开来维护。

首先，让我们建立一个包含你需要的所有规则的文本文件。没有什么特别的结构，不过是以空格 (或者制表符) 把字符串列成三列。你把它命名为 `rules.en`，“en” 是英语的意思。这些是英语名词复数的规则，你以后可以为其它语言添加规则文件。

例 17.15. `rules.en`

```
[sxz]$                  $               es
[^aeioudgkprt]h$        $               es
[^aeiou]y$              y$              ies
$                       $               s
```

现在来看看如何使用规则文件。

例 17.16. `plural5.py`

```
 import re
import string
def buildRule((pattern, search, replace)):
    return lambda word: re.search(pattern, word) and re.sub(search, replace, word)
def plural(noun, language='en'):
    lines = file('rules.%s' % language).readlines()
    patterns = map(string.split, lines)
    rules = map(buildRule, patterns)
    for rule in rules:
        result = rule(noun)
        if result: return result
```

[1]	在这里你还将使用闭合技术 (动态构建函数时使用函数外部定义的变量)，但是现在你把原来分开的匹配函数和规则应用函数合二为一 (你将在下一节中明了其原因)。你很快会看到，这与分别调用两个函数效果相同，只是调用的方法稍有不同。
[2]	咱们的 `plural` 函数现在接受的第二个参数是默认值为 `en` 的可选参数 `language` 。
[3]	你使用 `language` 参数命名一个文件，打开这个文件并读取其中的内容到一个列表。如果 `language` 是 `en` ，那么你将打开 `rules.en` 文件，读取全部内容，以其中的回车符作为分隔构建一个列表。文件的每一行将成为列表的一个元素。
[4]	如你所见，文件的每一行都有三个值，但是它们是以空白字符 (制表符或者空格符，这没什么区别) 分割。用 `string.split` 函数映射列表来创建一个每个元素都是三元素元组的新列表。因此，像 `[sxz]$ $ es` 这样的一行将被打碎并放入 `('[sxz]$', '$', 'es')` 这样的元组。这意味着 `patterns` 将最终变成元组列表的形式，就像第 4 阶段实打实编写的那样。
[5]	如果 `patterns` 是一个元组列表，那么 `rules` 就可以通过一个个调用 `buildRule` 动态地生成函数列表。调用 `buildRule(('[sxz]$', '$', 'es'))` 返回一个接受单参数 `word` 的函数。当返回的函数被调用，则将执行 `re.search('[sxz]$', word) and re.sub('$', 'es', word)` 。
[6]	因为你现在构建的是一个匹配和规则应用合一的函数，你需要分别调用它们。仅仅是调用函数，如果返回了内容，那么返回的便是复数；如果没有返回 (也就是返回了 `None`)，那么该规则未能匹配，就应该尝试其他规则。

这里的进步是你把复数规则完全分离到另外的文件中。不但这个文件可以独立于代码单独维护，而且你建立了一个命名规划使 `plural` 函数可以根据 `language` 参数使用不同的规则文件。

这里的缺陷是每次调用 `plural` 函数都需要去读取一次文件。我想我可以在整本书中都不使用 “留给读者去练习”，但是这里：为特定的语言规则文件建立一个缓存机制，并在调用期间规则文件改变时自动刷新留给读者作为练习。祝你顺利。

17.7. `plural.py` , 第 6 阶段

现在你已准备好探讨生成器 (Generator) 了。

例 17.17. `plural6.py`

```
 import re
def rules(language):
    for line in file('rules.%s' % language):
        pattern, search, replace = line.split()
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word)
def plural(noun, language='en'):
    for applyRule in rules(language):
        result = applyRule(noun)
        if result: return result
```

这里使用了被称作生成器的技术，我不打算在你看过一个简单例子之前试图解释它。

例 17.18. 介绍生成器

```
>>> def make_counter(x):
... print 'entering make_counter'
... while 1:
...         yield x
...         print 'incrementing x'
...         x = x + 1
...
>>> counter = make_counter(2)
>>> counter
<generator object at 0x001C9C10>
>>> counter.next()
entering make_counter
2
>>> counter.next()
incrementing x
3
>>> counter.next()
incrementing x
4
```

[1]	`make_counter` 中出现关键字 `yield` 意味着这不是一个普通的函数。它是一种每次生成一个值的特殊函数。你可以把它看成是一个可恢复函数。调用它会返回一个生成器，它可以返回 `x` 的连续值。
[2]	想要创建一个 `make_counter` 生成器的实例，只要像其它函数一样调用。注意这并没有真正执行函数代码。你可以分辨出这一点，因为 `make_counter` 的第一行是 `print` 语句，然而没有任何内容输出。
[3]	`make_counter` 函数返回一个生成器对象。
[4]	你第一次调用生成器对象的 `next()` 方法，将执行 `make_counter` 中的代码执行到第一个 `yield` 语句，然后返回生产 (yield) 出来的值。在本例中，这个值是 `2`，因为你是通过 `make_counter(2)` 来创建最初的生成器的。
[5]	不断调用生成器对象的 `next()` 将从你上次离开的位置重新开始 并继续下去直到你又一次遇到 `yield` 语句。接下来执行 `print` 语句来打印 `incrementing x`，然后执行 `x = x + 1` 语句来真正地增加。然后你进入 `while` 的又一次循环，你所做的第一件事是 `yield x`，返回目前的 `x` 值 (现在是3)。
[6]	第二次你调用 `counter.next()` 时，你又做一遍相同的事情，但是这次 `x` 是 `4`。如此继续。因为 `make_counter` 设置的是一个无限循环，理论上你可以永远这样继续下去，不断地递增并弹出 `x` 值。现在让我们看看生成器更具意义的应用。

例 17.19. 使用生成器替代递归

```
def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b
```

[1]	斐波纳契数列 (Fibonacci sequence) 是每个数都是前面两个数值和的一个数列。它从 `0` 和 `1` 开始，开始增长得很慢，但越来越快。开始这个数列你需要两个变量：`a` 从 `0` 开始，`b` 从 `1` 开始。
[2]	`a` 是数列的当前值，弹出它。
[3]	`b` 是数列的下一个数，把它赋值给 `a`，同时计算出 (`a+b`) 并赋值给 `b` 放在一边稍后使用。注意这是并行发生的，如果 `a` 是 `3`，`b` 是 `5`，那么 `a, b = b, a+b` 将会设置 `a` 为 `5` (`b` 的原值)，`b` 为 `8` (`a` 和 `b` 之和)。

这样你就有了生成连续的 Fibonacci 数的函数了。当然你也可以通过递归做到，但是这里的方法更加易读。并且也与 `for` 工作得很好。

例 17.20. `for` 循环中的生成器

```
>>> for n in fibonacci(1000):
... print n,
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

[1]	你可以在 `for` 循环中直接使用 `fibonacci` 这样的生成器。`for` 循环将会创建一个生成器对象并连续调用其 `next()` 方法获得值并赋予 `for` 循环变量 (`n`)。
[2]	每轮 `for` 循环 `n` 都从 `fibonacci` 的 `yield` 语句获得一个新的值。当 `fibonacci` 超出数字限定 (`a` 超过 `max` ，你在这里限定的是 `1000`) 很自然地退出 `for` 循环。

好了，让我们回到 `plural` 函数看看如何可以把它用起来。

例 17.21. 生成器生成动态函数

```
 def rules(language):
    for line in file('rules.%s' % language):
        pattern, search, replace = line.split()
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word)
def plural(noun, language='en'):
    for applyRule in rules(language):
        result = applyRule(noun)
        if result: return result
```

[1]	`for line in file(...)` 是从文件中一行行读取的通用方法，每次一行。它能正常工作是因为 *`file`* 实际上返回一个生成器，它的 `next()` 方法返回文件中的下一行。简直太酷了，光是想想就让我满头大汗。
[2]	这没有什么神奇之处。还记得规则文件的每一行都用空白分开三个值吗？所以 `line.split()` 返回一个三元素元组，你把这些值赋给了 3 个局部变量。
[3]	然后你不断地弹出。你弹出什么呢？一个使用 `lambda` 动态生成的函数，而这个函数实际上是一个闭合 (把本地变量 `pattern` ， `search` 和 `replace` 作为常量)。换句话说， `rules` 是一个弹出规则函数的生成器。
[4]	既然 `rules` 是一个生成器，你就可以在 `for` 循环中直接使用它。`for` 循环的第一轮你调用 `rules` 函数，打开规则文件，读取第一行，动态构建一个根据规则文件第一行匹配并应用规则的函数。`for` 循环的第二轮将会从上一轮 `rules` 中停下的位置 (`for line in file(...)` 循环内部) 读取规则文件的第二行，动态构建根据规则文件第二行匹配并应用规则的另一个函数。如此继续下去。

你在第 5 阶段得到的是什么？第 5 阶段中，你读取整个规则文件并在使用第一条规则之前构建一个所有规则组成的列表。现在有了生成器，你可以更舒适地做到这一切：你打开并读取第一条规则，根据它创建函数并使用之，如果它适用则根本不去读取规则文件剩下的内容，也不去建立另外的函数。

进一步阅读

- PEP 255 定义生成器。
- Python Cookbook 有生成器的例子。

17.8. 小结

这一章中我们探讨了几个不同的高级技术。它们并不都适用于任何情况。

你现在应该能自如应用如下技术：

- 应用正则表达式进行字符串替换。
- 将函数当作对象，把它们存于列表中，把它们赋值给变量，并通过变量来调用它们。
- 构建应用 `lambda` 的动态函数。
- 构建闭合，将外部变量作为常量构建动态函数。
- 构建生成器，进行逻辑递增操作并在每次调用时返回不同值的恢复执行函数。

抽象化，动态构建函数，构建闭合以及应用生成器能够使你的代码更加简单化、可读化、灵活化。你需要在简洁和功能实现两方面进行平衡。

第 18 章 性能优化

性能优化 (Performance tuning) 是一件多姿多彩的事情。Python 是一种解释性语言并不表示你不应该担心代码优化。但也不必太 担心。

18.1. 概览

由于代码优化过程中存在太多的不明确因素，以至于你很难清楚该从何入手。

让我们从这里开始：你真的确信你要这样做吗？ 你的代码真的那么差吗？值得花时间去优化它吗？在你的应用程序的生命周期中，与花费在等待一个远程数据库服务器，或是等待用户输入相比，运行这段代码将花费多少时间？

第二，你确信已经完成代码编写了吗？ 过早的优化就像是在一块半生不熟的蛋糕上撒糖霜。你花费了几小时、几天 (或更长) 时间来优化你的代码以提高性能，却发现它不能完成你希望它做的工作。那是浪费时间。

这并不是说代码优化毫无用处，但是你需要检查一下整个系统，并且确定把时间花在这上面是值得的。在优化代码上每花费一分钟，就意味着你少了增加新功能、编写文档或者陪你的孩子玩或者编写单元测试的一分钟。

哦，是的，单元测试。不必我说，在开始性能优化之前你需要一个完全的单元测试集。你最不需要的就是在乱动你的算法时引入新的问题。

谨记着这些忠告，让我们来看一些优化 Python 代码的技术。我们要研究的代码是 Soundex 算法的实现。Soundex 是一种 20 世纪在美国人口普查中归档姓氏的方法。它把听起来相似的姓氏归在一起，使得在即便错误拼写的情况下调查者仍能查找到。Soundex 今天仍然因差不多的原因被应用着，当然现在用计算机数据库服务器了。大部分的数据库服务器都有 Soundex 函数。

Soundex 算法有几个差别不大的变化版本。这是本章使用的：

1. 名字的第一个字母不变。
2. 根据特定的对照表，将剩下的字母转换为数字：
    - B、 F、 P 和 V 转换为 1。
    - C、 G、 J、 K、 Q、 S、 X 和 Z 转换为 2。
    - D 和 T 转换为 3。
    - L 转换为 4。
    - M 和 N 转换为 5。
    - R 转换为 6。
    - 所有其他字母转换为 9。
3. 去除连续重复。
4. 去除所有 9。
5. 如果结果都少于四个字符 (第一个字母加上后面的三位字符)，就以零补齐。
6. 如果结果超过四个字符，丢弃掉四位之后的字符。

比如，我的名字 `Pilgrim` 被转换为 P942695。没有连续重复，所以这一步不需要做。然后是去除 9，剩下 P4265。太长了，所以你把超出的字符丢弃，剩下 P426。

另一个例子：`Woo` 被转换为 W99，变成 W9，变成 W，然后以补零成为 W000。

这是 Soundex 函数的第一次尝试：

例 18.1. `soundex/stage1/soundex1a.py`

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
 import string, re
charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}
def soundex(source):
    "convert string to Soundex equivalent"
    # Soundex requirements:
    # source string must be at least 1 character
    # and must consist entirely of letters
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"
    # Soundex algorithm:
    # 1\. make first character uppercase
    source = source[0].upper() + source[1:]
    # 2\. translate all other characters to Soundex digits
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
    # 3\. remove consecutive duplicates
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    # 4\. remove all "9"s
    digits3 = re.sub('9', '', digits2)
    # 5\. pad end with "0"s to 4 characters
    while len(digits3) < 4:
        digits3 += "0"
    # 6\. return first 4 characters
    return digits3[:4]
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

进一步阅读

- Soundexing and Genealogy 给出了 Soundex 发展的年代表以及地域变化。

18.2. 使用 `timeit` 模块

关于 Python 代码优化你需要知道的最重要问题是，决不要自己编写计时函数。

为一个很短的代码计时都很复杂。处理器有多少时间用于运行这个代码？有什么在后台运行吗？每个现代计算机都在后台运行持续或者间歇的程序。小小的疏忽可能破坏你的百年大计，后台服务偶尔被 “唤醒” 在最后千分之一秒做一些像查收信件，连接计时通信服务器，检查应用程序更新，扫描病毒，查看是否有磁盘被插入光驱之类很有意义的事。在开始计时测试之前，把一切都关掉，断开网络的连接。再次确定一切都关上后关掉那些不断查看网络是否恢复的服务等等。

接下来是计时框架本身引入的变化因素。Python 解释器是否缓存了方法名的查找？是否缓存代码块的编译结果？正则表达式呢? 你的代码重复运行时有副作用吗？不要忘记，你的工作结果将以比秒更小的单位呈现，你的计时框架中的小错误将会带来不可挽回的结果扭曲。

Python 社区有句俗语：“Python 自己带着电池。” 别自己写计时框架。Python 2.3 具备一个叫做 `timeit` 的完美计时工具。

例 18.2. `timeit` 介绍

如果您还没有下载本书附带的样例程序, 可以 下载本程序和其他样例程序。

```
>>> import timeit
>>> t = timeit.Timer("soundex.soundex('Pilgrim')",
... "import soundex")
>>> t.timeit()
8.21683733547
>>> t.repeat(3, 2000000)
[16.48319309109, 16.46128984923, 16.44203948912]
```

[1]	`timeit` 模块定义了接受两个参数的 `Timer` 类。两个参数都是字符串。第一个参数是你要计时的语句，这里你计时的是以 `'Pilgrim'` 参数调用 Soundex 函数。传递给 `Timer` 的第二个参数是为第一个参数语句构建环境的导入语句。从内部讲，`timeit` 构建起一个独立的虚拟环境，手工地执行建立语句 (导入 `soundex` 模块)，然后手工地编译和执行被计时语句 (调用 Soundex 函数)。
[2]	只要有了 `Timer` 对象，最简单的事就是调用 `timeit()`，它调用你的函数一百万次并返回所耗费的秒数。
[3]	`Timer` 对象的另一个主要方法是 `repeat()`，它接受两个可选参数。第一个参数是重复整个测试的次数，第二个参数是每个测试中调用被计时语句的次数。两个参数都是可选的，它们的默认值分别是 `3` 和 `1000000`。`repeat()` 方法返回以秒记录的每个测试循环的耗时列表。

> 提示
> 你可以在命令行使用 `timeit` 模块来测试一个已存在的 Python 程序，而不需要修改代码。在 http://docs.python.org/lib/node396.html 查看文档中关于命令行选项的内容。

注意 `repeat()` 返回一个时间列表。由于 Python 计时器使用的处理器时间的微小变化 (或者那些你没办法根除的可恶的后台进程)，这些时间中几乎不可能出现重复。你的第一想法也许是说："让我们求平均值获得真实的数据。"

事实上，那几乎是确定错误的。你的代码或者 Python 解释器的变化可能缩短耗时，那些没办法去除的可恶后台进程或者其他 Python 解释器以外的因素也许令耗时延长。如果计时结果之间的差异超过百分之几，太多的可变因素使你没法相信结果，如果不是这样则可以取最小值而丢弃其他结果。

Python 有一个方便的 `min` 函数返回输入列表中的最小值：

```
>>> min(t.repeat(3, 1000000))
8.22203948912
```

> 提示
> `timeit` 模块只有在你知道哪段代码需要优化时使用。如果你有一个很大的 Python 程序并且不知道你的性能问题所在，查看 `hotshot` 模块。

18.3. 优化正则表达式

Soundex 函数的第一件事是检查输入是否是一个空字符串。怎样做是最好的方法？

如果你回答 “正则表达式”，坐在角落里反省你糟糕的直觉。正则表达式几乎永远不是最好的答案，而且应该被尽可能避开。这不仅仅是基于性能考虑，而是因为调试和维护都很困难，当然性能也是个原因。

这是 `soundex/stage1/soundex1a.py` 检查 `source` 是否全部由字母构成的一段代码，至少是一个字母 (而不是空字符串)：

```
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"
```

`soundex1a.py` 表现如何？为了方便，`__main__` 部分包含了一段代码：调用 `timeit` 模块，为三个不同名字分别建立测试，依次测试，并显示每个测试的最短耗时：

```
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

那么，应用正则表达式的 `soundex1a.py` 表现如何呢？

```
C:\samples\soundex\stage1>python soundex1a.py
Woo             W000 19.3356647283
Pilgrim         P426 24.0772053431
Flingjingwaller F452 35.0463220884
```

正如你预料，名字越长，算法耗时就越长。有几个工作可以令我们减小这个差距 (使函数对于长输入花费较短的相对时间) 但是算法的本质决定它不可能每次运行时间都相同。

另一点应铭记于心的是，我们测试的是有代表性的名字样本。`Woo` 是个被缩短到单字符并补零的小样本；`Pilgrim` 是个夹带着特别字符和忽略字符的平均长度的正常样本；`Flingjingwaller` 是一个包含连续重复字符并且特别长的样本。其它的测试可能同样有帮助，但它们已经很好地代表了不同的样本范围。

那么那个正则表达式如何呢？嗯，缺乏效率。因为这个表达式测试不止一个范围的字符 (`A-Z` 的大写范围和 `a-z` 的小写字母范围)，我们可以使用一个正则表达式的缩写语法。这便是 `soundex/stage1/soundex1b.py`：

```
    if not re.search('^[A-Za-z]+$', source):
        return "0000"
```

`timeit` 显示 `soundex1b.py` 比 `soundex1a.py` 稍微快一些，但是没什么令人激动的变化：

```
C:\samples\soundex\stage1>python soundex1b.py
Woo             W000 17.1361133887
Pilgrim         P426 21.8201693232
Flingjingwaller F452 32.7262294509
```

在 第 15.3 节 “重构” 中我们看到正则表达式可以被编译并在重用时以更快速度获得结果。因为这个正则表达式在函数中每次被调用时都不变化，我们可以编译它一次并使用被编译的版本。这便是 `soundex/stage1/soundex1c.py`：

```
isOnlyChars = re.compile('^[A-Za-z]+$').search
def soundex(source):
    if not isOnlyChars(source):
        return "0000"
```

`soundex1c.py` 中使用被编译的正则表达式产生了显著的提速：

```
C:\samples\soundex\stage1>python soundex1c.py
Woo             W000 14.5348347346
Pilgrim         P426 19.2784703084
Flingjingwaller F452 30.0893873383
```

但是这样的优化是正路吗？这里的逻辑很简单：输入 `source` 应该是非空，并且需要完全由字母构成。如果编写一个循环查看每个字符并且抛弃正则表达式，是否会更快些？

这便是 `soundex/stage1/soundex1d.py`：

```
    if not source:
        return "0000"
    for c in source:
        if not ('A' <= c <= 'Z') and not ('a' <= c <= 'z'):
            return "0000"
```

这个技术在 `soundex1d.py` 中恰好不及 编译后的正则表达式快 (尽管比使用未编译的正则表达式快[14])：

```
C:\samples\soundex\stage1>python soundex1d.py
Woo             W000 15.4065058548
Pilgrim         P426 22.2753567842
Flingjingwaller F452 37.5845122774
```

为什么 `soundex1d.py` 没能更快？答案来自 Python 的编译本质。正则表达式引擎以 C 语言编写，被编译后则能本能地在你的计算机上运行。另一方面，循环是以 Python 编写，要通过 Python 解释器。尽管循环相对简单，但没能简单到补偿花在代码解释上的时间。正则表达式

永远不是正确答案……但例外还是存在的。

恰巧 Python 提供了一个晦涩的字符串方法。你有理由不了解它，因为本书未曾提到它。这个方法便是 `isalpha()`，它检查一个字符串是否只包含字母。

这便是 `soundex/stage1/soundex1e.py`：

```
    if (not source) and (not source.isalpha()):
        return "0000"
```

在 `soundex1e.py` 中应用这个特殊方法我们能得到多少好处? 很多。

```
C:\samples\soundex\stage1>python soundex1e.py
Woo             W000 13.5069504644
Pilgrim         P426 18.2199394057
Flingjingwaller F452 28.9975225902
```

例 18.3. 目前为止最好的结果：`soundex/stage1/soundex1e.py`

```
 import string, re
charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}
def soundex(source):
    if (not source) and (not source.isalpha()):
        return "0000"
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

Footnotes

[14] 注意 `soundex1d.py` 在后两个测试点上都比 `soundex1b.py` 慢，这点与作者所说的矛盾。本章另还有多处出现了正文与测试结果矛盾的地方，每个地方都会用译注加以说明。这个 bug 将在下个版本中得到修正。——译注

18.4. 优化字典查找

Soundex 算法的第二步是依照特定规则将字符转换为数字。做到这点最好的方法是什么？

最明显的解决方案是定义一个以单字符为键并以所对应数字为值的字典，以字典查找每个字符。这便是 `soundex/stage1/soundex1e.py` 中使用的方法 (目前最好的结果)：

```
charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}
def soundex(source):
    # ... input check omitted for brevity ...
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
```

你已经为 `soundex1e.py` 计时，这便是其表现：

```
C:\samples\soundex\stage1>python soundex1c.py
Woo             W000 13.5069504644
Pilgrim         P426 18.2199394057
Flingjingwaller F452 28.9975225902
```

这段代码很直接，但它是最佳解决方案吗？为每个字符分别调用 `upper()` 看起来不是很有效率，为整个字符串调用 `upper()` 一次可能会好些。

然后是一砖一瓦地建立 `digits` 字符串。一砖一瓦的建造好像非常欠缺效率。在 Python 内部，解释器需要在循环的每一轮创建一个新的字符串，然后丢弃旧的。

但是，Python 擅长于列表。可以自动地将字符串作为列表来对待。而且使用 `join()` 方法可以很容易地将列表合并成字符串。

这便是 `soundex/stage2/soundex2a.py`，通过 `map` 和 `lambda` 把所有字母转换为数字：

```
def soundex(source):
    # ...
    source = source.upper()
    digits = source[0] + "".join(map(lambda c: charToSoundex[c], source[1:]))
```

太震惊了，`soundex2a.py` 并不快：

```
C:\samples\soundex\stage2>python soundex2a.py
Woo             W000 15.0097526362
Pilgrim         P426 19.254806407
Flingjingwaller F452 29.3790847719
```

匿名 `lambda` 函数的使用耗费掉了从以字符列表替代字符串争取来的时间。

`soundex/stage2/soundex2b.py` 使用了一个列表遍历来替代 `map` 和 `lambda`：

```
    source = source.upper()
    digits = source[0] + "".join([charToSoundex[c] for c in source[1:]])
```

在 `soundex2b.py` 中使用列表遍历比 `soundex2a.py` 中使用 `map` 和 `lambda` 快，但还没有最初的代码快 (`soundex1e.py` 中一砖一瓦的构建字符串[15])：

```
C:\samples\soundex\stage2>python soundex2b.py
Woo             W000 13.4221324219
Pilgrim         P426 16.4901234654
Flingjingwaller F452 25.8186157738
```

是时候从本质不同的方法来思考了。字典查找是一个普通目的实现工具。字典的键可以是任意长度的字符串 (或者很多其他数据类型) 但这里我们只和单字符键和 单字符值打交道。恰巧 Python 有处理这种情况的特别函数：`string.maketrans` 函数。

这便是 `soundex/stage2/soundex2c.py`：

```
allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
def soundex(source):
    # ...
    digits = source[0].upper() + source[1:].translate(charToSoundex)
```

这儿在干什么？`string.maketrans` 创建一个两个字符串间的翻译矩阵：第一参数和第二参数。就此而言，第一个参数是字符串 `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`，第二个参数是字符串 `9123912992245591262391929291239129922455912623919292`。看到其模式了？恰好与我们用冗长

的字典构建的模式相同。A 映射到 9，B 映射到 1，C 映射到 2 等等。但它不是一个字典。而是一个你可以通过字符串方法 `translate` 使用的特别数据结构。它根据 `string.maketrans` 定义的矩阵将每个字符翻译为对应的数字。

`timeit` 显示 `soundex2c.py` 比定义字典并对输入进行循环一砖一瓦地构建输出快很多：

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 11.437645008
Pilgrim         P426 13.2825062962
Flingjingwaller F452 18.5570110168
```

你不可能做得更多了。Python 有一个特殊函数，通过使用它做到了一个和你的工作差不多的事情。就用它并继续吧！

例 18.4. 目前的最佳结果：`soundex/stage2/soundex2c.py`

```
 import string, re
allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
def soundex(source):
    if (not source) or (not source.isalpha()):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

Footnotes

[15] 事实恰好相反，`soundex2b.py` 在每个点上都快于 `soundex1e.py` 。——译注

18.5. 优化列表操作

Soundex 算法的第三步是去除连续重复字符。怎样做是最佳方法？

这里是我们目前在 `soundex/stage2/soundex2c.py` 中的代码：

```
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
```

这里是 `soundex2c.py` 的性能表现：

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 11.437645008
Pilgrim         P426 13.2825062962
Flingjingwaller F452 18.5570110168
```

第一件事是考虑，考察在循环的每一轮都检查 `digits[-1]` 是否有效率。列表索引代价大吗？如果把上一个数字存在另外的变量中以便检查是否会获益？

这里的 `soundex/stage3/soundex3a.py` 将回答这个问题：

```
    digits2 = ''
    last_digit = ''
    for d in digits:
        if d != last_digit:
            digits2 += d
            last_digit = d
```

`soundex3a.py` 并不比 `soundex2c.py` 运行得快多少，而且甚至可能更会慢些 (差异还没有大到可以确信这一点)：

```
C:\samples\soundex\stage3>python soundex3a.py
Woo             W000 11.5346048171
Pilgrim         P426 13.3950636184
Flingjingwaller F452 18.6108927252
```

为什么 `soundex3a.py` 不更快呢？其实 Python 的索引功能恰恰很有效。重复使用 `digits2[-1]` 根本没什么问题。另一方面，手工保留上一个数字意味着我们每存储一个数字都要为两个 变量赋值，这便抹杀了我们避开索引查找所带来的微小好处。

让我们从本质上不同的方法来思考。如果可以把字符串当作字符列表来对待，那么使用列表遍历遍寻列表便成为可能。问题是代码需要使用列表中的上一个字符，而且使用列表遍历做到这一点并不容易。

但是，使用内建的 `range()` 函数创建一个索引数字构成的列表是可以的。使用这些索引数字一步步搜索列表并拿出与前面不同的字符。这样将使你得到一个字符串列表，使用字符串方法 `join()` 便可重建字符串。

这便是 `soundex/stage3/soundex3b.py` ：

```
    digits2 = "".join([digits[i] for i in range(len(digits))
                       if i == 0 or digits[i-1] != digits[i]])
```

这样快了吗？一个字，否。

```
C:\samples\soundex\stage3>python soundex3b.py
Woo             W000 14.2245271396
Pilgrim         P426 17.8337165757
Flingjingwaller F452 25.9954005327
```

有可能因为目前的这些方法都是 “字符串中心化” 的。Python 可以通过一个命令把一个字符串转化为一个字符列表： `list('abc')` 返回 `['a', 'b', 'c']` 。更进一步，列表可以被很快地就地 改变。与其一砖一瓦地建造一个新的列表 (或者字符串)，为什么不选择操作列表的元素呢？

这便是 `soundex/stage3/soundex3c.py` ，就地修改列表去除连续重复元素：

```
    digits = list(source[0].upper() + source[1:].translate(charToSoundex))
    i=0
    for item in digits:
        if item==digits[i]: continue
        i+=1
        digits[i]=item
    del digits[i+1:]
    digits2 = "".join(digits)
```

这比 `soundex3a.py` 或 `soundex3b.py` 快吗？不，实际上这是目前最慢的一种方法[16]：

```
C:\samples\soundex\stage3>python soundex3c.py
Woo             W000 14.1662554878
Pilgrim         P426 16.0397885765
Flingjingwaller F452 22.1789341942
```

我们在这儿除了试用了几种 “聪明” 的技术，根本没有什么进步。到目前为止最快的方法就是最直接的原始方法 (`soundex2c.py`)。有时候聪明未必有回报。

例 18.5. 目前的最佳结果： `soundex/stage2/soundex2c.py`

```
 import string, re
allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
def soundex(source):
    if (not source) or (not source.isalpha()):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

Footnotes

[16] `soundex3c.py` 比 `soundex3b.py` 快。——译注

18.6. 优化字符串操作

Soundex 算法的最后一步是对短结果补零和截短长结果。最佳的做法是什么？

这是目前在 `soundex/stage2/soundex2c.py` 中的做法：

```
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]
```

这里是 `soundex2c.py` 的表现：

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 12.6070768771
Pilgrim         P426 14.4033353401
Flingjingwaller F452 19.7774882003
```

思考的第一件事是以循环取代正则表达式。这里的代码来自 `soundex/stage4/soundex4a.py`：

```
    digits3 = ''
    for d in digits2:
        if d != '9':
            digits3 += d
```

`soundex4a.py` 快了吗？是的：

```
C:\samples\soundex\stage4>python soundex4a.py
Woo             W000 6.62865531792
Pilgrim         P426 9.02247576158
Flingjingwaller F452 13.6328416042
```

但是，等一下。一个从字符串去除字符的循环？我们可以用一个简单的字符串方法做到。这便是 `soundex/stage4/soundex4b.py`：

```
    digits3 = digits2.replace('9', '')
```

`soundex4b.py` 快了吗？这是个有趣的问题，它取决输入值：

```
C:\samples\soundex\stage4>python soundex4b.py
Woo             W000 6.75477414029
Pilgrim         P426 7.56652144337
Flingjingwaller F452 10.8727729362
```

`soundex4b.py` 中的字符串方法对于大多数名字比循环快，但是对于短小的情况 (很短的名字) 却比 `soundex4a.py` 略微慢些。性能优化并不总是一致的，对于一个情况快些，却可能对另外一些情况慢些。就此而言，大多数情况将会从改变中获益，所以就改吧，但是别忘了原则。

最后仍很重要的是，让我们检测算法的最后两步：以零补齐短结果和截短超过四字符的长结果。你在 `soundex4b.py` 中看到的代码就是做这个工作的，但是太没效率了。看一下 `soundex/stage4/soundex4c.py` 找出原因：

```
    digits3 += '000'
    return digits3[:4]
```

我们为什么需要一个 `while` 循环来补齐结果？我们早就知道我们需要把结果截成四字符，并且我们知道我们已经有了至少一个字符 (直接从 `source` 中拿过来的起始字符)。这意味着我们可以仅仅在输出的结尾添加三个零，然后截断它。不要害怕重新理解问题，从不太一样的角度看问题可以获得简单的解决方案。

我们丢弃 `while` 循环后从 `soundex4c.py` 中获得怎样的速度？太明显了：

```
C:\samples\soundex\stage4>python soundex4c.py
Woo             W000 4.89129791636
Pilgrim         P426 7.30642134685
Flingjingwaller F452 10.689832367
```

最后，还有一件事可以令这三行运行得更快：你可以把它们合并为一行。看一眼 `soundex/stage4/soundex4d.py`：

```
    return (digits2.replace('9', '') + '000')[:4]
```

在 `soundex4d.py` 中把所有代码放在一行可以比 `soundex4c.py` 稍微快那么一点：

```
C:\samples\soundex\stage4>python soundex4d.py
Woo             W000 4.93624105857
Pilgrim         P426 7.19747593619
Flingjingwaller F452 10.5490700634
```

它非常难懂，而且优化也不明显。这值得吗？我希望你有很好的见解。性能并不是一切。你在优化方面的努力应该与程序的可读性和可维护性相平衡。

18.7. 小结

这一章展示了性能优化的几个重要方面，这里是就 Python 而言，但它们却普遍适用。

- 如果你要在正则表达式和编写循环间抉择，选择正则表达式。正则表达式因其是以 C 语言编译的可以本能地在你的计算机上运行，你的循环却以 Python 编写需要通过 Python 解释器运行。
- 如果你需要在正则表达式和字符串方法间抉择，选择字符串方法。它们都是以 C 编译的，所以选取简单的。
- 字典查找的通常应用很快，但是 `string.maketrans` 之类的特殊函数和 `isalpha()` 之类的字符串方法更快。如果 Python 有定制方法给你用，就使它吧！
- 别太聪明了。有时一些明显的算法是最快的。
- 不要太迷恋性能优化，性能并不是一切。

最后一点太重要了，这章中你令这个程序提速三倍并且令百万次的调用节省 20 秒。太棒了！现在思考一下：在那百万次的函数调用中，有多少秒花在周边应用程序等待数据库连接？花在磁盘输入/输出上？花在等待用户输入上？不要在过度优化算法上花时间，从而忽略了其它地方可以做的明显改进。开发你编写运行良好的 Python 代码的直觉，如果发现明显的失误则修正它们，并不对其它部分过分操作。

附录 A. 进一步阅读

第 1 章 安装 Python

第 2 章 第一个 Python 程序

- 2.3. 文档化函数
  - PEP 257 定义了 `doc string` 规范。
  - *Python Style Guide* 讨论了如何编写一个好的 `doc string` 。
  - *Python Tutorial* 讨论了在 `doc string` 中如何使用空白。
- 2.4.2. 何谓对象？
  - *Python Reference Manual* 确切解释了在 Python 中万物皆对象的含义，因为有些书生气十足的人，喜欢花时间讨论这类的问题。
  - eff-bot 总结了 Python 对象.
- 2.5. 代码缩进
  - *Python Reference Manual* 讨论了交叉缩进问题，并且演示了各种各样的缩进错误。
  - *Python Style Guide* 讨论了良好的缩进风格。
- 2.6. 测试模块
  - *Python Reference Manual* 讨论了导入模块的底层细节。

第 3 章 内置数据类型

- 3.1.3. 从 dictionary 中删除元素
  - *How to Think Like a Computer Scientist* 讲授了 dictionary 和如何使用 dictionary 模拟稀疏矩阵。
  - Python Knowledge Base 有许多使用 dictionary 的示例代码。
  - Python Cookbook 讨论了如何通过 key 对 dictionary 的值进行排序。
  - *Python Library Reference* 总结了所有的 dictionary 方法。
- 3.2.5. 使用 list 的运算符
  - *How to Think Like a Computer Scientist* 讲述了list，并且重点讲述了如何把 list 作为函数参数传递。
  - *Python Tutorial* 展示了如何把 list 作为堆栈和队列使用。
  - Python Knowledge Base 回答了有关 list 的常见问题并且有许多使用 list 的示例代码。
  - *Python Library Reference* 总结了所有的 list 方法。
- 3.3. Tuple 介绍
  - *How to Think Like a Computer Scientist* 讲解了 tuple 并且展示了如何连接 tuple。
  - Python Knowledge Base 展示了如何对一个 tuple 排序。
  - *Python Tutorial* 展示了如何定义一个只包含一个元素的 tuple。
- 3.4.2. 一次赋多值

- *Python Reference Manual* 展示了什么时候可以忽略续行符和什么时候您需要使用续行符的例子。
- *How to Think Like a Computer Scientist* 演示了如何使用多变量赋值来交换两个变量的值。

- 3.5. 格式化字符串
  - *Python Library Reference* 总结了所有字符串格式化所使用的格式符。
  - *Effective AWK Programming*Top) 讨论了所有的格式符Control+Letters)和高级字符串格式化技术，如指定宽度，精度和0填充Format+Modifiers)。
- 3.6. 映射 list
  - *Python Tutorial* 讨论了另一种方法来映射 list：使用内置的 `map` 函数。
  - *Python Tutorial* 展示了如何对嵌套 list 的 list 进行解析。
- 3.7. 连接 list 与分割字符串
  - Python Knowledge Base 回答了关于字符串的常见问题，并且有许多使用字符串的例子代码。
  - *Python Library Reference* 总结了所有字符串方法。
  - *Python Library Reference* 提供了 `string` 模块的文档。
  - *The Whole Python FAQ* 解释了为什么 `join` 是字符串方法而不是 list 方法。

第 4 章 自省的威力

- 4.2. 使用可选参数和命名参数
  - *Python Tutorial* 确切地讨论了何时、如何进行缺省参数赋值，这都和缺省值是一个 list 还是一个具有副作用的表达式有关。
- 4.3.3. 内置函数
  - *Python Library Reference* 对所有的内置函数和所有的内置异常都进行了文档化。
- 4.5. 过滤列表
  - *Python Tutorial* 讨论了使用内置 `filter` 函数过滤列表的另一种方式。
- 4.6.1. 使用 and-or 技巧
  - Python Cookbook 讨论了其它的 `and-or` 技巧。
- 4.7.1. 真实世界中的 lambda 函数
  - Python Knowledge Base 讨论了使用 `lambda` 来间接调用函数。
  - *Python Tutorial* 演示了如何从一个 `lambda` 函数内部访问外部变量。(PEP 227 解释了在 Python 的未来版本中将如何变化。)
  - *The Whole Python FAQ* 有关于令人模糊的使用 `lambda` 单行语句的例子。

第 5 章 对象和面向对象

- 5.2. 使用 from module import 导入模块
  - eff-bot 有更多关于 `import _module_` *vs.* `from _module_ import` 的论述。
  - *Python Tutorial* 讨论了高级的导入技术，包括 `from _module_ import *`。
- 5.3.2. 了解何时去使用 self 和 **init**
  - *Learning to Program* 有优雅的类的介绍。

- *How to Think Like a Computer Scientist* 展示了如何使用类来实现复合数据类型模型。
- *Python Tutorial* 深入考虑了类、名字空间和继承。
- Python Knowledge Base 回答了关于类的常见问题。

- 5.4.1. 垃圾回收
  - *Python Library Reference* 总结了像 `__class__` 之类的内置属性。
  - *Python Library Reference* 提供了 `gc` 模块的文档，此模块给予你对 Python 的垃圾回收的底层控制权。
- 5.5. 探索 UserDict：一个封装类
  - *Python Library Reference* 提供了 `UserDict` 模块 和 `copy` 模块 的文档。
- 5.7. 高级专用类方法
  - *Python Reference Manual* 提供了所有专用类方法的文档。
- 5.9. 私有函数
  - *Python Tutorial* 讨论了私有变量的内部工作方式。

第 6 章 异常和文件处理

- 6.1.1. 为其他用途使用异常
  - *Python Tutorial* 讨论了异常，包括定义和引发你自己的异常，以及一次处理多个异常。
  - *Python Library Reference* 总结了所有内置异常。
  - *Python Library Reference* 提供了 getpass 模块的文档。
  - *Python Library Reference* 提供了 `traceback` 模块 的文档，这个模块在异常引发之后，提供了底层的对异常属性的处理。
  - *Python Reference Manual* 讨论了 `try...except` 块 的内部工作方式。
- 6.2.4. 写入文件
  - *Python Tutorial* 讨论了文件的读取和写入，包括如何将一个文件一次一行地读到 list 中。
  - eff-bot 讨论了各种各样读取文件方法 的效率和性能。
  - Python Knowledge Base 回答了关于文件的常见问题。
  - *Python Library Reference* 总结了所有文件对象模块。
- 6.4. 使用 sys.modules
  - *Python Tutorial* 讨论了缺省参数到底在什么时候和是如何计算的。
  - *Python Library Reference* 提供了 `sys` 模块的文档。
- 6.5. 与目录共事
  - Python Knowledge Base 回答了关于 `os` 模块的问题。
  - *Python Library Reference* 提供了 `os` 模块和 `os.path` 模块的文档。

第 7 章 正则表达式

- 7.6. 个案研究：解析电话号码
  - Regular Expression HOWTO 讲解正则表达式和如何在 Python 中使用正则表达式。

- *Python Library Reference* 概述了 `re` module。

第 8 章 HTML 处理

- 8.4. BaseHTMLProcessor.py 介绍
  - W3C 讨论了字符和实体引用。
  - *Python Library Reference* 解答了您的怀疑，即 `htmlentitydefs` 模块的确名符其实。
- 8.9. 全部放在一起
  - 您可能会认为我的服务端脚本编程的想法是开玩笑。在我发现这个基于 web 的方言转换器之前，的确是这样想的。不幸的是，看不到它的源代码。

第 9 章 XML 处理

- 9.4. Unicode
  - Unicode.org 是 unicode 标准的主页，包含了一个简要的技术简介。
  - Unicode 教程有更多关于如何使用 Python unicode 函数的例子，包括甚至在并不真的需要时如何将 unicode 强制转换为 ASCII。
  - PEP 263 涉及了何时、如何在你的 `.py` 文件中定义字符编码的更多细节。

第 10 章 脚本和流

第 11 章 HTTP Web 服务

- 11.1. 概览
  - Paul Prescod 认为纯 HTTP web 服务是 Internet 的未来。

第 12 章 SOAP Web 服务

- 12.1. 概览
  - http://www.xmethods.net/ 是一个访问 SOAP web 服务的公共知识库。
  - SOAP 规范相当可读，如果你喜欢这类东西的话。
- 12.8. SOAP 网络服务故障排除
  - New developments for SOAPpy 一步步连接到另一个不名副其实的 SOAP 服务。

第 13 章 单元测试

- 13.1. 罗马数字程序介绍 II
  - 这个站点 有关于罗马数字更多的内容，包括罗马人如何使用罗马数字的迷人 历史 (简言之：充满偶然性和反复无常)。
- 13.3. romantest.py 介绍
  - PyUnit 主页 对于使用 `unittest` 框架 以及本章没能涵盖的高级特性有深入的讨论。
  - PyUnit FAQ 解释了 为什么测试用例要和被测试代码分开存放 。
  - *Python Library Reference* 总结了 `unittest` 模块。
  - ExtremeProgramming.org 讨论 你为什么需要编写单元测试。

    - The Portland Pattern Repository 有一个持续的 单元测试 讨论，包括了一个 标准的定义，为什么你需要 首先开发单元测试代码 以及另外一些深层次 案例。

第 14 章 测试优先编程

第 15 章 重构

- 15.5. 小结
    - XProgramming.com 有多种语言的 单元测试框架 的下载链接。

第 16 章 函数编程

第 17 章 动态函数

- 17.7. plural.py, 第 6 阶段
    - PEP 255 定义生成器。
    - Python Cookbook 有生成器的例子。

第 18 章 性能优化

- 18.1. 概览
    - Soundexing and Genealogy 给出了 Soundex 发展的年代表以及地域变化。

附录 B. 五分钟回顾

第 1 章 安装 Python

- 1.1. 哪一种 Python 适合您？

  学习 Python 的第一件事就是安装，不是吗？

- 1.2. Windows 上的 Python

  在 Windows 上，安装 Python 有两种选择。

- 1.3. Mac OS X 上的 Python

  在 Mac OS X 上，对于安装 Python 有两种选择：安装或不安装。您可能想要安装它。

- 1.4. Mac OS 9 上的 Python

  Mac OS 9 上没有预装任何版本的 Python，安装相对简单，只有一种选择。

- 1.5. RedHat Linux 上的 Python

  在 http://www.python.org/ftp/python/ 选择列出的最新的版本号, 然后选择 其中的 `rpms/` 目录下载最新的 Python RPM 包。 使用 **rpm** 命令进行安装，操作如下所示:

- 1.6. Debian GNU/Linux 上的 Python

  如果您运行在 Debian GNU/Linux 上，安装 Python 需要使用 **apt** 命令。

- 1.7. 从源代码安装 Python

  如果您宁愿从源码创建，可以从 http://www.python.org/ftp/python/下载 Python 的源代码。选择最新的版本，下载 `.tgz` 文件，执行通常的 **`configure`**，**`make`**，**`make install`** 步骤。

- 1.8. 使用 Python 的交互 Shell

  既然我们已经安装了 Python，那么我们运行的这个交互shell是什么东西呢？

- 1.9. 小结

  您现在应该已经安装了一个可以工作的 Python 版本了。

第 2 章 第一个 Python 程序

- 2.1. 概览

  这是一个完整的、可执行的 Python 程序。

- 2.2. 函数声明

  与其它大多数语言一样 Python 有函数，但是它没有像 C++ 一样的独立的头文件；或者像 Pascal 一样的分离的 `interface` / `implementation` 段。在需要函数时，像下面这样声明即可：

- 2.3. 文档化函数

  可以通过给出一个 `doc string` (文档字符串) 来文档化一个 Python 函数。

- 2.4. 万物皆对象

  在 Python 中，函数同其它东西一样也是对象。

- 2.5. 代码缩进

  Python 函数没有明显的 `begin` 和 `end`，没有标明函数的开始和结束的花括号。唯一的分隔符是一个冒号 (`:`)，接着代码本身是缩进的。

- 2.6. 测试模块

  所有的 Python 模块都是对象，并且有几个有用的属性。您可以使用这些属性方便地测试您所编写的模块。下面是一个使用 `if` `__name__` 的技巧。

第 3 章 内置数据类型

- 3.1. Dictionary 介绍

  Dictionary 是 Python 的内置数据类型之一，它定义了键和值之间一对一的关系。

- 3.2. List 介绍

  List 是 Python 中使用最频繁的数据类型。如果您对 list 仅有的经验就是在 Visual Basic 中的数组或 Powerbuilder 中的数据存储，那么就打起精神学习 Python 的 list 吧。

- 3.3. Tuple 介绍

  Tuple 是不可变的 list。一旦创建了一个 tuple，就不能以任何方式改变它。

- 3.4. 变量声明

  Python 与大多数其它语言一样有局部变量和全局变量之分，但是它没有明显的变量声明。变量通过首次赋值产生，当超出作用范围时自动消亡。

- 3.5. 格式化字符串

  Python 支持格式化字符串的输出 。尽管这样可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 `%s` 的字符串中。

- 3.6. 映射 list

Python 的强大特性之一是其对 list 的解析，它提供一种紧凑的方法，可以通过对 list 中的每个元素应用一个函数，从而将一个 list 映射为另一个 list。

- 3.7. 连接 list 与分割字符串

  您有了一个形如 `_key_=_value_` 的 key-value 对 list，并且想将它们合成为单个字符串。为了将任意包含字符串的 list 连接成单个字符串，可以使用字符串对象的 `join` 方法。

- 3.8. 小结

  现在 `odbchelper.py` 程序和它的输出结果都应该非常清楚了。

第 4 章 自省的威力

- 4.1. 概览

  下面是一个完整可运行的 Python 程序。大概看一下这段程序，你应该可以理解不少了。用数字标出的行阐述了 第 2 章 第一个 *Python* 程序 中涉及的一些概念。如果剩下来的代码看起来有点奇怪，不用担心，通过阅读本章你将会理解所有这些。

- 4.2. 使用可选参数和命名参数

  Python 允许函数参数有缺省值；如果调用函数时不使用参数，参数将获得它的缺省值。此外，通过使用命名参数还可以以任意顺序指定参数。SQL Server Transact/SQL 中的存储过程也可以做到这些；如果你是脚本高手，你可以略过这部分。

- 4.3. 使用 type、str、dir 和其它内置函数

  Python 有小部分相当有用的内置函数。除这些函数之外，其它所有的函数都被分到了各个模块中。其实这是一个非常明智的设计策略，避免了核心语言变得像其它脚本语言一样臃肿 (咳 咳，Visual Basic)。

- 4.4. 通过 getattr 获取对象引用

  你已经知道 Python 函数是对象。你不知道的是，使用 `getattr` 函数，可以得到一个直到运行时才知道名称的函数的引用。

- 4.5. 过滤列表

  如你所知，Python 具有通过列表解析 (第 3.6 节 “映射 list”) 将列表映射到其它列表的强大能力。这种能力同过滤机制结合使用，使列表中的有些元素被映射的同时跳过另外一些元素。

- 4.6. and 和 or 的特殊性质

  在Python 中，`and` 和 `or` 执行布尔逻辑演算，如你所期待的一样。但是它们并不返回布尔值，而是返回它们实际进行比较的值之一。

- 4.7. 使用 lambda 函数

Python 支持一种有趣的语法，它允许你快速定义单行的最小函数。这些叫做 `lambda` 的函数，是从 Lisp 借用来的，可以用在任何需要函数的地方。

- 4.8. 全部放在一起

  最后一行代码是唯一还没有解释过的，它完成全部的工作。但是现在工作已经简单了，因为所需要的每件事都已经按照需求建立好了。所有的多米诺骨牌已经就位，到了将它们推倒的时候了。

- 4.9. 小结

  `apihelper.py` 程序和它的输出现在应该非常清晰了。

第 5 章 对象和面向对象

- 5.1. 概览

  下面是一个完整的，可运行的 Python 程序。请阅读模块、类和函数的 `doc string`s，可以大概了解这个程序所做的事情和工作情况。像平时一样，不用担心你不理解的东西，这就是本章其它部分将告诉你的内容。

- 5.2. 使用 from module import 导入模块

  Python 有两种导入模块的方法。两种都有用，你应该知道什么时候使用哪一种方法。一种方法，`import _module_`，你已经在第 2.4 节 “万物皆对象”看过了。另一种方法完成同样的事情，但是它与第一种有着细微但重要的区别。

- 5.3. 类的定义

  Python 是完全面向对象的：你可以定义自已的类，从自已的或内置的类继承，然后从你定义的类创建实例。

- 5.4. 类的实例化

  在 Python 中对类进行实例化很直接。要对类进行实例化，只要调用类 (就好像它是一个函数)，传入定义在 `__init__` 方法中的参数。返回值将是新创建的对象。

- 5.5. 探索 UserDict：一个封装类

  如你所见，`FileInfo` 是一个有着像字典一样的行为方式的类。为了进一步揭示这一点，让我们看一看在 `UserDict` 模块中的 `UserDict` 类，它是我们的 `FileInfo` 类的父类。它没有什么特别的，也是用 Python 写的，并且保存在一个 `.py` 文件里，就像我们其他的代码。特别之处在于，它保存在你的 Python 安装目录的 `lib` 目录下。

- 5.6. 专用类方法

  除了普通的类方法，Python 类还可以定义专用方法。专用方法是在特殊情况下或当使用特别语法时由 Python 替你调用的，而不是在代码中直接调用 (像普通的方法那样)。

- 5.7. 高级专用类方法

  除了 `__getitem__` 和 `__setitem__` 之外 Python 还有更多的专用函数。某些可以让你模拟出你甚至可能不知道的功能。

- 5.8. 类属性介绍

  你已经知道了数据属性，它们是被一个特定的类实例所拥有的变量。Python 也支持类属性，它们是由类本身所拥有的。

- 5.9. 私有函数

  与大多数的语言不同，一个 Python 函数，方法，或属性是私有还是公有，完全取决于它的名字。

- 5.10. 小结

  实打实的对象把戏到此为止。你将在 第 12 章 中看到一个真实世界应用程序的专有类方法，它使用 `getattr` 创建一个到远程 Web 服务的代理。

第 6 章 异常和文件处理

- 6.1. 异常处理

  与许多面向对象语言一样，Python 具有异常处理，通过使用 `try...except` 块来实现。

- 6.2. 与文件对象共事

  Python 有一个内置函数，`open`，用来打开在磁盘上的文件。`open` 返回一个文件对象，它拥有一些方法和属性，可以得到被打开文件的信息，以及对被打开文件进行操作。

- 6.3. for 循环

  与其它大多数语言一样，Python 也拥有 `for` 循环。你到现在还未曾看到它们的唯一原因就是，Python 在其它太多的方面表现出色，通常你不需要它们。

- 6.4. 使用 sys.modules

  与其它任何 Python 的东西一样，模块也是对象。只要导入了，总可以用全局 dictionary `` `sys `` .modules` 来得到一个模块的引用。

- 6.5. 与目录共事

  `os.path` 模块有几个操作文件和目录的函数。这里，我们看看如何操作路径名和列出一个目录的内容。

- 6.6. 全部放在一起

再一次，所有的多米诺骨牌都放好了。我们已经看过每行代码是如何工作的了。现在往回走一步，看一下放在一起是怎么样的。

- 6.7. 小结

  在 第 5 章 介绍的 `fileinfo.py` 程序现在应该完全理解了。

第 7 章 正则表达式

- 7.1. 概览

  如果你要解决的问题利用字符串函数能够完成，你应该使用它们。它们快速、简单且容易阅读，而快速、简单、可读性强的代码可以说出很多好处。但是，如果你发现你使用了许多不同的字符串函数和 `if` 语句来处理一个特殊情况，或者你组合使用了 `split` 、 `join` 等函数而导致用一种奇怪的甚至读不下去的方式理解列表，此时，你也许需要转到正则表达式了。

- 7.2. 个案研究：街道地址

  这一系列的例子是由我几年前日常工作中的现实问题启发而来的，当时我需要从一个老化系统中导出街道地址，在将它们导入新的系统之前，进行清理和标准化。(看，我不是只将这些东西堆到一起，它有实际的用处。)这个例子展示我如何处理这个问题。

- 7.3. 个案研究：罗马字母

  你可能经常看到罗马数字，即使你没有意识到它们。你可能曾经在老电影或者电视中看到它们 (“版权所有 `MCMXLVI` ” 而不是 “版权所有 `1946` ”)，或者在某图书馆或某大学的贡献墙上看到它们 (“成立于 `MDCCCLXXXVIII` ”而不是“成立于 `1888` ”)。你也可能在某些文献的大纲或者目录上看到它们。这是一个表示数字的系统，它实际上能够追溯到远古的罗马帝国 (因此而得名)。

- 7.4. 使用 {n,m} 语法

  在前面的章节，你处理了相同字符可以重复三次的情况。在正则表达式中，有另外一个方式来表达这种情况，并且能提高代码的可读性。首先看看我们在前面的例子中使用的方法。

- 7.5. 松散正则表达式

  迄今为止，你只是处理过被我称之为“紧凑”类型的正则表达式。正如你曾看到的，它们难以阅读，即使你清楚正则表达式的含义，你也不能保证六个月以后你还能理解它。你真正所需的就是利用内联文档 (inline documentation)。

- 7.6. 个案研究：解析电话号码

迄今为止，你主要是匹配整个模式，不论是匹配上，还是没有匹配上。但是正则表达式还有比这更为强大的功能。当一个模式确实 匹配上时，你可以获取模式中特定的片断，你可以发现具体匹配的位置。

- 7.7. 小结

  这只是正则表达式能够完成工作的很少一部分。换句话说，即使你现在备受打击，相信我，你也不是什么也没见过了。

第 8 章 HTML 处理

- 8.1. 概览

  我经常在 comp.lang.python 上看到关于如下的问题：“怎么才能从我的 HTML 文档中列出所有的 [头|图像|链接] 呢？”“怎么才能 [分析|解释|munge] 我的 HTML 文档的文本，但是又要保留标记呢？”“怎么才能一次给我所有的 HTML 标记 [增加|删除|加引号] 属性呢？”本章将回答所有这些问题。

- 8.2. sgmllib.py 介绍

  HTML 处理分成三步：将 HTML 分解成它的组成片段，对片段进行加工，接着将片段再重新合成 HTML。第一步是通过 `sgmllib.py` 来完成的，它是标准 Python 库的一部分。

- 8.3. 从 HTML 文档中提取数据

  为了从 HTML 文档中提取数据，将 `SGMLParser` 类进行子类化，然后对想要捕捉的标记或实体定义方法。

- 8.4. BaseHTMLProcessor.py 介绍

  `SGMLParser` 自身不会产生任何结果。它只是分析，分析，再分析，对于它找到的有趣的东西会调用相应的一个方法，但是这些方法什么都不做。`SGMLParser` 是一个 HTML 消费者 *(consumer)*：它接收 HTML，将其分解成小的、结构化的小块。正如您所看到的，在前一节中，您可以定义 `SGMLParser` 的子类，它可以捕捉特别标记和生成有用的东西，如一个网页中所有链接的一个列表。现在我们将沿着这条路更深一步。我们要定义一个可以捕捉 `SGMLParser` 所丢出来的所有东西的一个类，接着重建整个 HTML 文档。用技术术语来说，这个类将是一个 HTML 生产者 *(producer)*。

- 8.5. locals 和 globals

  我们先偏离一下 HTML 处理的主题，讨论一下 Python 如何处理变量。Python 有两个内置的函数，`locals` 和 `globals`，它们提供了基于 dictionary 的访问局部和全局变量的方式。

- 8.6. 基于 dictionary 的字符串格式化

  有另外一种字符串格式化的形式，它使用 dictionary 而不是值的 tuple。

- 8.7. 给属性值加引号

  在 comp.lang.python 上的一个常见问题是 “我有一些 HTML 文档，属性值没有用引号括起来，并且我想将它们全部括起来，我怎么才能实现它呢？” [7] (一般这种事情的出现是由于一个项目经理加入到一个大的项目中来，而他又抱着 HTML 是一种标记语言的教条，要求所有的页面必须能够通过 HTML 校验器的验证。而属性值没有被引号括起来是一种常见的对 HTML 规范的违反。) 不管什么原因，未括起来的属性值通过将 HTML 送进 `BaseHTMLProcessor` 可以容易地修复。

- 8.8. dialect.py 介绍

  `Dialectizer` 是 `BaseHTMLProcessor` 的简单 (和拙劣) 的派生类。它通过一系列的替换对文本块进行了处理，但是它确保在 `` <pre>`...`</pre> `` 块之间的任何东西不被修改地通过。

- 8.9. 全部放在一起

  到了将迄今为止我们已经学过并用得不错的东西放在一起的时候了。我希望您专心些。

- 8.10. 小结

  Python 向您提供了一个强大工具， `sgmllib.py` ，可以通过将 HTML 结构转变为一种对象模型来进行处理。可以以许多不同的方式来使用这个工具。

第 9 章 XML 处理

- 9.1. 概览

  处理 XML 有两种基本的方式。一种叫做 SAX (“Simple API for XML”)，它的工作方式是，一次读出一点 XML 内容，然后对发现的每一个元素调用一个方法。(如果你读了 第 8 章 *HTML* 处理，这应该听起来很熟悉，因为这是 `sgmllib` 工作的方式。) 另一种方式叫做 DOM (“Document Object Model”)，它的工作方式是，一次性读入整个 XML 文档，然后使用 Python 类创建一个内部表示形式 (以树结构进行连接)。Python 拥有这两种解析方式的标准模块，但是本章只涉及 DOM。

- 9.2. 包

  实际上解析一个 XML 文档是很简单的：只要一行代码。但是，在你接触那行代码前，需要暂时岔开一下，讨论一下包。

- 9.3. XML 解析

  正如我说的，实际解析一个 XML 文档是非常简单的：只要一行代码。从这里出发到哪儿去就是你自己的事了。

- 9.4. Unicode

Unicode 是一个系统，用来表示世界上所有不同语言的字符。当 Python 解析一个 XML 文档时，所有的数据都是以unicode的形式保存在内存中的。

- 9.5. 搜索元素

  通过一步步访问每一个节点的方式遍历 XML 文档可能很乏味。如果你正在寻找些特别的东西，又恰恰它们深深埋入了你的 XML 文档，有个捷径让你可以快速找到它：`getElementsByTagName` 。

- 9.6. 访问元素属性

  XML 元素可以有一个或者多个属性，只要已经解析了一个 XML 文档，访问它们就太简单了。

- 9.7. Segue

  以上就是 XML 的核心内容。下一章将使用相同的示例程序，但是焦点在于能使程序更加灵活的其它方面：使用输入流处理，使用 `getattr` 进行方法分发，并使用命令行标识允许用户重新配置程序而无需修改代码。

第 10 章 脚本和流

- 10.1. 抽象输入源

  Python 的最强大力量之一是它的动态绑定，而动态绑定最强大的用法之一是类文件*(file-like)*对象。

- 10.2. 标准输入、输出和错误

  UNIX 用户已经对标准输入、标准输出和标准错误的概念非常熟悉了。这一节是为其他不熟悉的人准备的。

- 10.3. 查询缓冲节点

  `kgp.py` 使用了多种技巧，在你进行 XML 处理时，它们或许能派上用场。第一个就是，利用输入文档的结构稳定特征来构建节点缓冲。

- 10.4. 查找节点的直接子节点

  解析 XML 文档时，另一个有用的己技巧是查找某个特定元素的所有直接子元素。例如，在语法文件中，一个 `ref` 元素可以有数个 `p` 元素，其中每一个都可以包含很多东西，包括其他的 `p` 元素。你只要查找作为 `ref` 孩子的 `p` 元素，不用查找其他 `p` 元素的孩子 `p` 元素。

- 10.5. 根据节点类型创建不同的处理器

第三个有用的 XML 处理技巧是将你的代码基于节点类型和元素名称分散到逻辑函数中。解析后的 XML 文档是由各种类型的节点组成的，每一个都是通过 Python 对象表示的。文档本身的根层次通过一个 `Document` 对象表示。`Document` 还包含了一个或多个 `Element` 对象 (表示 XML 标记)，其中的每一个可以包含其它的 `Element` 对象、`Text` 对象 (表示文本)，或者 `Comment` 对象 (表示内嵌注释)。使用 Python 编写分离各个节点类型逻辑的分发器非常容易。

- 10.6. 处理命令行参数

  Python 完全支持创建在命令行运行的程序，也支持通过命令行参数和短长样式来指定各种选项。这些并非是 XML 特定的，但是这样的脚本可以充分使用命令行处理，看来是时候提一下它了。

- 10.7. 全部放在一起

  你已经了解很多基础的东西。让我们回来看看所有片段是如何整合到一起的。

- 10.8. 小结

  Python 带有解析和操作 XML 文档非常强大的库。`minidom` 接收一个 XML 文件并将其解析为 Python 对象，并提供了对任意元素的随机访问。进一步，本章展示了如何利用 Python 创建一个“真实”独立的命令行脚本，连同命令行标志、命令行参数、错误处理，甚至从前一个程序的管道接收输入的能力。

第 11 章 HTTP Web 服务

- 11.1. 概览

  在讲解如何下载 web 页和如何从 URL 解析 XML时，你已经学习了关于 HTML 处理和 XML 处理，接下来让我们来更全面地探讨有关 HTTP web 服务的主题。

- 11.2. 避免通过 HTTP 重复地获取数据

  假如说你想用 HTTP 下载资源，例如一个 Atom feed 汇聚。你不仅仅想下载一次；而是想一次又一次地下载它，如每小时一次，从提供 news feed 的站点获得最新的消息。让我们首先用一种直接而原始的方法来实现它，然后看看如何改进它。

- 11.3. HTTP 的特性

  这里有五个你必须关注的 HTTP 重要特性。

- 11.4. 调试 HTTP web 服务

  首先，让我们开启 Python HTTP 库的调试特性并查看网络线路上的传输过程。这对本章的全部内容都很有用，因为你将添加越来越多的特性。

- 11.5. 设置 User-Agent

改善你的 HTTP web 服务客户端的第一步就是用 `User-Agent` 适当地鉴别你自己。为了做到这一点，你需要远离基本的 `urllib` 而深入到 `urllib2` 。

- 11.6. 处理 Last-Modified 和 ETag

  既然你知道如何在你的 web 服务请求中添加自定义的 HTTP 头信息，接下来看看如何添加 `Last-Modified` 和 `ETag` 头信息的支持。

- 11.7. 处理重定向

  你可以使用两种不同的自定义 URL 处理器来处理永久重定向和临时重定向。

- 11.8. 处理压缩数据

  你要支持的最后一个重要的 HTTP 特性是压缩。许多 web 服务具有发送压缩数据的能力，这可以将网络线路上传输的大量数据消减 60% 以上。这尤其适用于 XML web 服务，因为 XML 数据 的压缩率可以很高。

- 11.9. 全部放在一起

  你已经看到了构造一个智能的 HTTP web 客户端的所有片断。现在让我们看看如何将它们整合到一起。

- 11.10. 小结

  `openanything.py` 及其函数现在可以完美地工作了。

第 12 章 SOAP Web 服务

- 12.1. 概览

  你用 Google，对吧？它是一个很流行的搜索引擎。你是否希望能以程序化的方式访问 Google 的搜索结果呢？现在你能做到了。下面是一个用 Python 搜索 Google 的程序。

- 12.2. 安装 SOAP 库

  与本书中的其他代码不同，本章依赖的库不是 Python 预安装的。

- 12.3. 步入 SOAP

  调用远程函数是 SOAP 的核心功能。有很多提供公开 SOAP 访问的服务器提供用于展示的简单功能。

- 12.4. SOAP 网络服务查错

  SOAP 提供了一个很方便的方法用以查看背后的情形。

- 12.5. WSDL 介绍

`SOAPProxy` 类本地方法调用并透明地转向到远程 SOAP 方法。正如你所看到的，这是很多的工作，`SOAPProxy` 快速和透明地完成他们。它没有做到的是提供方法自省的手段。

- 12.6. 以 WSDL 进行 SOAP 内省

  就像网络服务舞台上的所有事物，WSDL 也经历了一个充满明争暗斗而且漫长多变的历史。我不打算讲述这段令我伤心的历史。还有一些其他的标准提供相同的支持，但 WSDL 还是胜出，所以我们还是来学习一下如何使用它。

- 12.7. 搜索 Google

  让我们回到这章开始时你看到的那段代码，获得比当前气温更有价值和令人振奋的信息。

- 12.8. SOAP 网络服务故障排除

  是的，SOAP 网络服务的世界中也不总是欢乐和阳光。有时候也会有故障。

- 12.9. 小结

  SOAP 网络服务是很复杂的，雄心勃勃的它试图涵盖网络服务的很多不同应用。这一章我们接触了它的一个简单应用。

第 13 章 单元测试

- 13.1. 罗马数字程序介绍 II

  在前面的章节中，通过阅读代码，你迅速“深入”，以最快的速度理解了各个程序。既然你已对 Python 有了一定的了解，那么接下来让我们看看程序开发之前 的工作。

- 13.2. 深入

  现在你已经定义了你的转换程序所应有的功能，下面一步会有点儿出乎你的意料：你将要开发一个测试组件 (test suite) 来测试你未来的函数以确保它们工作正常。没错：你将为还未开发的程序开发测试代码。

- 13.3. romantest.py 介绍

  这是将被开发并保存为 `roman.py` 的罗马数字转换程序的完整测试组件 (test suite)。很难立刻看出它们是如何协同工作的，似乎所有类或者方法之间都没有关系。这是有原因的，而且你很快就会明了。

- 13.4. 正面测试 (Testing for success)

  单元测试的基础是构建独立的测试用例 (test case)。一个测试用例只回答一个关于被测试代码的问题。

- 13.5. 负面测试 (Testing for failure)

使用有效输入确保函数成功通过测试还不够，你还需要测试无效输入导致函数失败的情形。但并不是任何失败都可以，必须如你预期地失败。

- 13.6. 完备性检测 (Testing for sanity)

  你经常会发现一组代码中包含互逆的转换函数，一个把 A 转换为 B ，另一个把 B 转换为 A。在这种情况下，创建“完备性检测”可以使你在由 A 转 B 再转 A 的过程中不会出现丢失精度或取整等错误。

第 14 章 测试优先编程

- 14.1. roman.py, 第 1 阶段

  到目前为止，单元测试已经完成，是时候开始编写被单元测试测试的代码了。你将分阶段地完成这个工作，因此开始时所有的单元测试都是失败的，但在逐步完成 `roman.py` 的同时你会看到它们一个个地通过测试。

- 14.2. roman.py, 第 2 阶段

  现在你有了 `roman` 模块的大概框架，到了开始写代码以通过测试的时候了。

- 14.3. roman.py, 第 3 阶段

  现在 `toRoman` 对于有效的输入 (`1` 到 `3999` 整数) 已能正确工作，是正确处理那些无效输入 (任何其他输入) 的时候了。

- 14.4. roman.py, 第 4 阶段

  现在 `toRoman` 完成了，是开始编写 `fromRoman` 的时候了。感谢那个将每个罗马数字和对应整数关连的完美数据结构，这个工作不比 `toRoman` 函数复杂。

- 14.5. roman.py, 第 5 阶段

  现在 `fromRoman` 对于有效输入能够正常工作了，是揭开最后一个谜底的时候了：使它正常工作于无效输入的情况下。这意味着要找出一个方法检查一个字符串是不是有效的罗马数字。这比 `toRoman` 中验证有效的数字输入困难，但是你可以使用一个强大的工具：正则表达式。

第 15 章 重构

- 15.1. 处理 bugs

  尽管你很努力地编写全面的单元测试，但是 bug 还是会出现。我所说的 “bug” 是什么呢？Bug 是你还没有编写的测试用例。

- 15.2. 应对需求变化

尽管你竭尽努力地分析你的客户，并点灯熬油地提炼出精确的需求，但需求还是会是不断变化。大部分客户在看到产品前不知道他们想要什么。即便知道，也不擅于精确表述出他们的有效需求。即便能表述出来，他们在下一个版本一定会要求更多的功能。因此你需要做好更新测试用例的准备以应对需求的改变。

- 15.3. 重构

  全面的单元测试带来的最大好处不是你的全部测试用例最终通过时的成就感；也不是被责怪破坏了别人的代码时能够证明 自己的自信。最大的好处是单元测试给了你自由去无情地重构。

- 15.4. 后记

  聪明的读者在学习前一节时想得会更深入一层。现在写的这个程序中最令人头痛的性能负担是正则表达式，但它是必需的，因为没有其它方法来识别罗马数字。但是，它们只有 5000 个，为什么不一次性地构建一个查询表来读取？不必用正则表达式凸现了这个主意的好处。你建立了整数到罗马数字查询表的时候，罗马数字到整数的逆向查询表也构建了。

- 15.5. 小结

  单元测试是一个强大的概念，使用得当的话既可以减少维护成本又可以增加长期项目的灵活性。同样重要的是要意识到单元测试并不是“灵丹妙药”，也不是“银弹”。编写好的测试用例很困难，保持其更新更需要磨练 (特别是当顾客对修复严重的 Bug 大呼小叫之时)。单元测试不是其它形式测试的替代品，比如说功能性测试、集成测试以及可用性测试。但它切实可行且功效明显，一旦相识，你会反问为什么以往没有应用它。

第 16 章 函数编程

- 16.1. 概览

  在 第 13 章 单元测试 中，你学会了单元测试的哲学。在 第 14 章 测试优先编程 中你步入了 Python 基本的单元测试操作，在 第 15 章 重构 部分，你看到单元测试如何令大规模重构变得容易。本章将在这些程序样例的基础上，集中关注于超越单元测试本身的更高级的 Python 特有技术。

- 16.2. 找到路径

  从命令行运行 Python 代码时，知道所运行代码在磁盘上的存储位置有时候是有必要的。

- 16.3. 重识列表过滤

  你已经熟识了应用列表解析来过滤列表。这里介绍的是达到相同效果的另一种令很多人感觉清晰的实现方法。

- 16.4. 重识列表映射

你对使用列表解析映射列表的做法已经熟知。另一种方法可以完成同样的工作：使用内建 `map` 函数。它的工作机理和 `filter` 函数类似。

- 16.5. 数据中心思想编程

  现在的你，可能正抓耳挠腮地狠想，为什么这样比使用 `for` 循环和直接调用函数好。这是一个非常好的问题。通常这是一个程序观问题。使用 `map` 和 `filter` 强迫你围绕数据进行思考。

- 16.6. 动态导入模块

  好了，大道理谈够了。让我们谈谈动态导入模块吧。

- 16.7. 全部放在一起

  你已经学习了足够的知识，现在来分析本章样例代码的前七行：读取一个目录并从中导入选定的模块。

- 16.8. 小结

  `regression.py` 程序及其输出到现在应该很清楚了。

第 17 章 动态函数

- 17.1. 概览

  我想谈谈名词复数。还有，返回其它函数的函数，高级的正则表达式和生成器 (Generator)。生成器是 Python 2.3 新引入的。但首先还是让我们先来谈谈如何生成名词复数。

- 17.2. plural.py, 第 1 阶段

  你所针对的单词 (至少在英语中) 是字符串和字符。你还需要规则来找出不同的字符 (字母) 组合，并对它们进行不同的操作。这听起来像是正则表达式的工作。

- 17.3. plural.py, 第 2 阶段

  现在你将增加一个抽象过程。你从定义一个规则列表开始：如果这样，就做那个，否则判断下一规则。让我们暂时将程序一部分复杂化以便使另一部分简单化。

- 17.4. plural.py, 第 3 阶段

  将每个匹配和规则应用分别制作成函数没有必要。你从来不会直接调用它们：你把它们定义于 `rules` 列表之中并从那里调用它们。让我们隐去它们的函数名而抓住规则定义的主线。

- 17.5. plural.py, 第 4 阶段

  让我们精炼出代码中的重复之处，以便更容易地定义新规则。

- 17.6. plural.py, 第 5 阶段

  你已经精炼了所有重复代码，也尽可能地把复数规则提炼到定义一个字符串列表。接下来的步骤是把这些字符串提出来放在另外的文件中，从而可以和使用它们的代码分开来维护。

- 17.7. plural.py, 第 6 阶段

  现在你已准备好探讨生成器 (Generator) 了。

- 17.8. 小结

  这一章中我们探讨了几个不同的高级技术。它们并不都适用于任何情况。

第 18 章 性能优化

- 18.1. 概览

  由于代码优化过程中存在太多的不明确因素，以至于你很难清楚该从何入手。

- 18.2. 使用 timeit 模块

  关于 Python 代码优化你需要知道的最重要问题是，决不要自己编写计时函数。

- 18.3. 优化正则表达式

  Soundex 函数的第一件事是检查输入是否是一个空字符串。怎样做是最好的方法？

- 18.4. 优化字典查找

  Soundex 算法的第二步是依照特定规则将字符转换为数字。做到这点最好的方法是什么？

- 18.5. 优化列表操作

  Soundex 算法的第三步是去除连续重复字符。怎样做是最佳方法？

- 18.6. 优化字符串操作

  Soundex 算法的最后一步是对短结果补零和截短长结果。最佳的做法是什么？

- 18.7. 小结

  这一章展示了性能优化的几个重要方面，这里是就 Python 而言，但它们却普遍适用。

附录 C. 技巧和窍门

第 1 章 安装 Python

第 2 章 第一个 Python 程序

- 2.1. 概览

> 提示
> 在 Windows 的 ActivePython IDE 中，可以选择 File->Run... (**Ctrl-R**) 来运行 Python 程序。输出结果将显示在交互窗口中。
>
> 提示
> 在 Mac OS 的 Python IDE 中，可以选择 Python->Run window... (**Cmd-R**) 来运行 Python 程序，但首先要设置一个重要的选项。在 IDE 中打开 `.py` 模块，点击窗口右上角的黑色三角，弹出这个模块的选项菜单，然后将 Run as **main** 选中。 这个设置是同模块一同保存的，所以对于每个模块您都需要这样做。
>
> 提示
> 在 UNIX 兼容的操作系统中 (包括 Mac OS X)，可以通过命令行：`python` **odbchelper.py**`` 运行模块。

- 2.2. 函数声明

> 注意
> 在 Visual Basic 中，函数 (有返回值) 以 `function` 开始，而子程序 (无返回值) 以 `sub` 开始。在 Python 中没有子程序。只有函数，所有的函数都有返回值 (尽管可能为 `None`)，并且所有的函数都以 `def` 开始。
>
> 注意
> 在 Java、C++ 和其他静态类型语言中，必须要指定函数返回值和每个函数参数的数据类型。在 Python 中，永远也不需要明确指定任何东西的数据类型。Python 会根据赋给它的值在内部将其数据类型记录下来。

- 2.3. 文档化函数

> 注意
> 三重引号也是一种定义既包含单引号又包含双引号的字符串的简单方法，就像 Perl 中的 `qq/.../` 。
>
> 注意
> 许多 Python IDE 使用 `doc string` 来提供上下文敏感的文档信息，所以当键入一个函数名时，它的 `doc string` 显示为一个工具提示。这一点可以说非常有用，但是它的好坏取决于您书写的 `doc string` 的好坏。

- 2.4. 万物皆对象

> 注意
> 在 Python 中的 `import` 就像 Perl 中的 `require`。`import` 一个 Python 模块后，您就可以使用 `_module_._function_` 来访问它的函数；`require` 一个 Perl 模块后，您就可以使用 `_module_::_function_` 来访问它的函数。

- 2.5. 代码缩进

> 注意
> Python 使用硬回车来分割语句，冒号和缩进来分割代码块。C++ 和 Java 使用分号来分割语句，花括号来分割代码块。

- 2.6. 测试模块

> 注意
> 与 C 一样，Python 使用 `==` 做比较，使用 `=` 做赋值。与 C 不一样，Python 不支持行内赋值，所以不会出现想要进行比较却意外地出现赋值的情况。
>
> 提示
> 在 MacPython 上，需要一个额外的步聚来使得 `if` `__name__` 技巧有效。点击窗口右上角的黑色三角，弹出模块的属性菜单，确认 Run as **main** 被选中。

第 3 章 内置数据类型

- 3.1. Dictionary 介绍

> 注意
> Python 中的 dictionary 就像 Perl 中的 hash (哈希数组)。在 Perl 中，存储哈希值的变量总是以 `%` 字符开始；在 Python 中，变量可以任意取名，并且 Python 在内部会记录下其数据类型。
>
> 注意
> Python 中的 dictionary 像 Java 中的 `Hashtable` 类的实例。
>
> 注意
> Python 中的 dictionary 像 Visual Basic 中的 `Scripting.Dictionary` 对象的实例。

- 3.1.2. Dictionary 的修改

> 注意
> Dictionary 没有元素顺序的概念。说元素 “顺序乱了” 是不正确的，它们只是序偶的简单排列。这是一个重要的特性，它会在您想要以一种特定的，可重现的顺序 (像以 key 的字母表顺序) 存取 dictionary 元素的时候骚扰您。有一些实现这些要求的方法，它们只是没有加到 dictionary 中去。

- 3.2. List 介绍

> 注意
> Python 的 list 如同 Perl 中的数组。在 Perl 中，用来保存数组的变量总是以 `@` 字符开始；在 Python 中，变量可以任意取名，并且 Python 在内部会记录下其数据类型。
>
> 注意
> Python 中的 list 更像 Java 中的数组 (您可以简单地这样理解，但 Python 中的 list 远比 Java 中的数组强大)。一个更好的类比是 `ArrayList` 类，它可以保存任意对象，并且可以在增加新元素时动态扩展。

- 3.2.3. 在 list 中搜索

> 注意
> 在 2.2.1 版本之前，Python 没有单独的布尔数据类型。为了弥补这个缺陷，Python 在布尔环境 (如 `if` 语句) 中几乎接受所有东西，遵循下面的规则：
>
> - `0` 为 false; 其它所有数值皆为 true。
> - 空串 (`""`) 为 false; 其它所有字符串皆为 true。
> - 空 list (`[]`) 为 false; 其它所有 list 皆为 true。
> - 空 tuple (`()`) 为 false; 其它所有 tuple 皆为 true。
> - 空 dictionary (`{}`) 为 false; 其它所有 dictionary 皆为 true。
>
> 这些规则仍然适用于 Python 2.2.1 及其后续版本，但现在您也可以使用真正的布尔值，它的值或者为 `True` 或者为 `False`。请注意第一个字母是大写的；这些值如同在 Python 中的其它东西一样都是大小写敏感的。

- 3.3. Tuple 介绍

> 注意
> Tuple 可以转换成 list，反之亦然。内置的 `tuple` 函数接收一个 list，并返回一个有着相同元素的 tuple。而 `list` 函数接收一个 tuple 返回一个 list。从效果上看，`tuple` 冻结一个 list，而 `list` 解冻一个 tuple。

- 3.4. 变量声明

> 注意
> 当一条命令用续行符 (“`\`”) 分割成多行时，后续的行可以以任何方式缩进，此时 Python 通常的严格的缩进规则无需遵守。如果您的 Python IDE 自由对后续行进行了缩进，您应该把它当成是缺省处理，除非您有特别的原因不这么做。

- 3.5. 格式化字符串

> 注意
> 在 Python 中，字符串格式化使用与 C 中 `sprintf` 函数一样的语法。

- 3.7. 连接 list 与分割字符串

> 小心
> `join` 只能用于元素是字符串的 list；它不进行任何的强制类型转换。连接一个存在一个或多个非字符串元素的 list 将引发一个异常。

> 提示
> `_anystring_.`split`(_delimiter_, 1)` 是一个有用的技术，在您想要搜索一个子串，然后分别处理理字符前半部分 (即 list 中第一个元素) 和后半部分 (即 list 中第二个元素) 时，使用这个技术。

第 4 章 自省的威力

- 4.2. 使用可选参数和命名参数

> 注意
> 调用函数时唯一必须做的事情就是为每一个必备参数指定值 (以某种方式)；以何种具体的方式和顺序都取决于你。

- 4.3.3. 内置函数

> 注意
> Python 提供了很多出色的参考手册，你应该好好地精读一下所有 Python 提供的必备模块。对于其它大部分语言，你会发现自己要常常回头参考手册或者 man 页来提醒自己如何使用这些模块，但是 Python 不同于此，它很大程度上是自文档化的。

- 4.7. 使用 lambda 函数

> 注意
> `lambda` 函数是一种风格问题。不一定非要使用它们；任何能够使用它们的地方，都可以定义一个单独的普通函数来进行替换。我将它们用在需要封装特殊的、非重用代码上，避免令我的代码充斥着大量单行函数。

- 4.8. 全部放在一起

> 注意
> 在 SQL 中，你必须使用 `IS NULL` 代替 `= NULL` 进行 null 值比较。在 Python，你可以使用 `== None` 或者 `is None` 进行比较，但是 `is None` 更快。

第 5 章 对象和面向对象

- 5.2. 使用 from module import 导入模块

> 注意
> Python 中的 `from _module_ import *` 像 Perl 中的 `use _module_` ；Python 中的 `import _module_` 像 Perl 中的 `require _module_` 。
>
> 注意
> Python 中的 `from _module_ import *` 像 Java 中的 `import _module_.*` ；Python 中的 `import _module_` 像 Java 中的 `import _module_` 。
>
> 小心
> 尽量少用 `from module import *` ，因为判定一个特殊的函数或属性是从哪来的有些困难，并且会造成调试和重构都更困难。

- 5.3. 类的定义

> 注意
> 在 Python 中的 `pass` 语句就像 Java 或 C 中的大括号空集 (`{}`)。
>
> 注意
> 在 Python 中，类的基类只是简单地列在类名后面的小括号里。不像在 Java 中有一个特殊的 `extends` 关键字。

- 5.3.1. 初始化并开始类编码

> 注意
> 习惯上，任何 Python 类方法的第一个参数 (对当前实例的引用) 都叫做 `self` 。这个参数扮演着 C++ 或 Java 中的保留字 `this` 的角色，但 `self` 在 Python 中并不是一个保留字，它只是一个命名习惯。虽然如此，也请除了 `self` 之外不要使用其它的名字，这是一个非常坚固的习惯。

- 5.3.2. 了解何时去使用 self 和 **init**

> 注意
> `__init__` 方法是可选的，但是一旦你定义了，就必须记得显示调用父类的 `__init__` 方法 (如果它定义了的话)。这样更是正确的：无论何时子类想扩展父类的行为，后代方法必须在适当的时机，使用适当的参数，显式调用父类方法。

- 5.4. 类的实例化

> 注意
> 在 Python 中，创建类的实例只要调用一个类，仿佛它是一个函数就行了。不像 C++ 或 Java 有一个明确的 `new` 操作符。

- 5.5. 探索 UserDict：一个封装类

> 提示
> 在 Windows 下的 ActivePython IDE 中，你可以快速打开在你的库路径中的任何模块，使用 File->Locate... (**Ctrl-L**)。
>
> 注意
> Java 和 Powerbuilder 支持通过参数列表的重载，也就是 一个类可以有同名的多个方法，但这些方法或者是参数个数不同，或者是参数的类型不同。其它语言 (最明显如 PL/SQL) 甚至支持通过参数名的重载，也就是 一个类可以有同名的多个方法，这些方法有相同类型，相同个数的参数，但参数名不同。Python 两种都不支持，总之是没有任何形式的函数重载。一个 `__init__` 方法就是一个 `__init__` 方法，不管它有什么样的参数。每个类只能有一个 `__init__` 方法，并且如果一个子类拥有一个 `__init__` 方法，它总是 覆盖父类的 `__init__` 方法，甚至子类可以用不同的参数列表来定义它。
>
> 注意
> Python 的原作者 Guido 是这样解释方法覆盖的："子类可以覆盖父类中的方法。因为方法没有特殊的优先级设置，父类中的一个方法在调用同类中的另一方法时，可能其实调用用到的却是一个子类中覆盖父类同名方法的方法。 (C++ 程序员可能会这样想：所有的 Python 方法都是虚函数。)"如果你不明白 (它令我颇感困惑)，不必在意。我想我要跳过它。[3]
>
> 小心
> 应该总是在 `__init__` 方法中给一个实例的所有数据属性赋予一个初始值。这样做将会节省你在后面调试的时间，不必为捕捉因使用未初始化 (也就是不存在) 的属性而导致的 `AttributeError` 异常费时费力。
>
> 注意
> 在 Python 2.2 之前的版本中，你不可以直接子类化字符串、列表以及字典之类的内建数据类型。作为补偿，Python 提供封装类来模拟内建数据类型的行为，比如：`UserString`、`UserList` 和 `UserDict`。通过混合使用普通和特殊方法，`UserDict` 类能十分出色地模仿字典。在 Python 2.2 和其后的版本中，你可以直接从 `dict` 内建数据类型继承。本书 `fileinfo_fromdict.py` 中有这方面的一个例子。

- 5.6.1. 获得和设置数据项

> 注意
> 当在一个类中存取数据属性时，你需要限定属性名：`self._attribute_`。当调用类中的其它方法时，你属要限定方法名：`self._method_`。

- 5.7. 高级专用类方法

> 注意
> 在 Java 中，通过使用 `str1 == str2` 可以确定两个字符串变量是否指向同一块物理内存位置。这叫做对象同一性，在 Python 中写为 `str1 is str2`。在 Java 中要比较两个字符串值，你要使用 `str1.equals(str2)`；在 Python 中，你要使用 `str1 == str2`。某些 Java 程序员，他们已经被教授得认为，正是因为在 Java 中 `==` 是通过同一性而不是值进行比较，所以世界才会更美好。这些人要接受 Python 的这个“严重缺失”可能要花些时间。
>
> 注意
> 其它的面向对象语言仅让你定义一个对象的物理模型 (“这个对象有 `GetLength` 方法”)，而 Python 的专用类方法像 `__len__` 允许你定义一个对象的逻辑模型 (“这个对象有一个长度”)。

- 5.8. 类属性介绍

> 注意
> 在 Java 中，静态变量 (在 Python 中叫类属性) 和实例变量 (在 Python 中叫数据属性) 两者都是紧跟在类定义之后定义的 (一个有 `static` 关键字，一个没有)。在 Python 中，只有类属性可以定义在这里，数据属性定义在 `__init__` 方法中。
>
> 注意
> 在 Python 中没有常量。如果你试图努力的话什么都可以改变。这一点满足 Python 的核心原则之一：坏的行为应该被克服而不是被取缔。如果你真正想改变 `None` 的值，也可以做到，但当无法调试的时候别来找我。

- 5.9. 私有函数

> 注意
> 在 Python 中，所有的专用方法 (像 `__setitem__`](specialclassmethods.html#fileinfo.specialmethods.setitem.example "例 5.13. ***setitem*** 专用方法")) 和内置属性 (像 [`__doc`) 遵守一个标准的命名习惯：开始和结束都有两个下划线。不要对你自已的方法和属性用这种方法命名；到最后，它只会把你 (或其它人) 搞乱。

第 6 章 异常和文件处理

- 6.1. 异常处理

> 注意
> Python 使用 `try...except` 来处理异常，使用 `raise` 来引发异常。Java 和 C++ 使用 `try...catch` 来处理异常，使用 `throw` 来引发异常。

- 6.5. 与目录共事

> 注意
> 只要有可能，你就应该使用在 `os` 和 `os.path` 中的函数进行文件、目录和路径的操作。这些模块是对平台相关模块的封装模块，所以像 `os.path.split` 这样的函数可以工作在 UNIX、Windows、Mac OS 和 Python 所支持的任一种平台上。

第 7 章 正则表达式

- 7.4. 使用 {n,m} 语法

> 注意
> 没有一个轻松的方法来确定两个正则表达式是否等价。你能采用的最好的办法就是列出很多的测试样例，确定这两个正则表达式对所有的相关输入都有相同的输出。在本书后面的章节，将更多地讨论如何编写测试样例。

第 8 章 HTML 处理

- 8.2. sgmllib.py 介绍

> 重要
> Python 2.0 存在一个 bug，即 `SGMLParser` 完全不能识别声明 (`handle_decl` 永远不会调用)，这就意味着 `DOCTYPE` 被静静地忽略掉了。这个错误在 Python 2.1 中改正了。
>
> 提示
> 在 Windows 下的 ActivePython IDE 中，您可以在 “Run script” 对话框中指定命令行参数。用空格将多个参数分开。

- 8.4. BaseHTMLProcessor.py 介绍

> 重要
> HTML 规范要求所有非 HTML (像客户端的 JavaScript) 必须包括在 HTML 注释中，但不是所有的页面都是这么做的 (而且所有的最新的浏览器也都容许不这样做) 。 `BaseHTMLProcessor` 不允许这样，如果脚本嵌入得不正确，它将被当作 HTML 一样进行分析。例如，如果脚本包含了小于和等于号， `SGMLParser` 可能会错误地认为找到了标记和属性。 `SGMLParser` 总是把标记名和属性名转换成小写，这样可能破坏了脚本，并且 `BaseHTMLProcessor` 总是用双引号来将属性封闭起来 (尽管原始的 HTML 文档可能使用单引号或没有引号) ，这样必然会破坏脚本。应该总是将您的客户端脚本放在 HTML 注释中进行保护。

- 8.5. locals 和 globals

> 重要
> Python 2.2 引入了一种略有不同但重要的改变，它会影响名字空间的搜索顺序：嵌套的作用域。 在 Python 2.2 版本之前，当您在一个嵌套函数或 `lambda` 函数中引用一个变量时，Python 会在当前 (嵌套的或 `lambda`) 函数的名字空间中搜索，然后在模块的名字空间。Python 2.2 将只在当前 (嵌套的或 `lambda`) 函数的名字空间中搜索，然后是在父函数的名字空间 中搜索，接着是模块的名字空间中搜索。Python 2.1 可 以两种方式工作，缺省地，按 Python 2.0 的方式工作。但是您可以把下面一行代码增加到您的模块头部，使您的模块工作起来像 Python 2.2 的方式：
>
> ```
> from __future__ import nested_scopes
> ```
>
> 注意
> 使用 `locals` 和 `globals` 函数，通过提供变量的字符串名字您可以动态地得到任何变量的值。这种方法提供了这样的功能：`getattr` 函数允许您通过提供函数的字符串名来动态地访问任意的函数。

- 8.6. 基于 dictionary 的字符串格式化

> 重要
> 使用 `locals` 来应用基于 dictionary 的字符串格式化是一种方便的作法，它可以使复杂的字符串格式化表达式更易读。但它需要花费一定的代价。在调用 `locals` 方面有一点性能上的问题，这是由于 `locals` 创建了局部名字空间的一个拷贝引起的。

第 9 章 XML 处理

- 9.2. 包

> 注意
> 一个包是一个其中带有特殊文件 `__init__.py` 的目录。`__init__.py` 文件定义了包的属性和方法。其实它可以什么也不定义；可以只是一个空文件，但是必须要存在。如果 `__init__.py` 不存在，这个目录就仅仅是一个目录，而不是一个包，它就不能被导入或者包含其它的模块和嵌套包。

- 9.6. 访问元素属性

> 注意
> 这部分由于某个含义重叠的术语可能让人有点糊涂。在一个 XML 文档中，元素可以有属性，而 Python 对象也有属性。当你解析一个 XML 文档时，你得到了一组 Python 对象，它们代表 XML 文档中的所有片段，同时有些 Python 对象代表 XML 元素的属性。但是表示 (XML) 属性的 (Python) 对象也有 (Python) 属性，它们用于访问对象表示的 (XML) 属性。我告诉过你它让人糊涂。我会公开提出关于如何更明显地区分这些不同的建议。
>
> 注意
> 类似于字典，一个 XML 元素的属性没有顺序。属性可以以某种顺序偶然 列在最初的 XML 文档中，而在 XML 文档解析为 Python 对象时，`Attr` 对象以某种顺序偶然列出，这些顺序都是任意的，没有任何特别的含义。你应该总是使用名称来访问单个属性，就像字典的键一样。

第 10 章 脚本和流

第 11 章 HTTP Web 服务

- 11.6. 处理 Last-Modified 和 ETag

> 注意
> 在这些例子中，HTTP 服务器同时支持 `Last-Modified` 和 `ETag` 头信息，但并非所有的服务器皆如此。作为一个 web 服务的客户端，你应该为支持两种头信息做准备，但是你的程序也应该为服务器仅支持其中一种头信息或两种头信息都不支持而做准备。

第 12 章 SOAP Web 服务

第 13 章 单元测试

- 13.2. 深入

> 注意
> Python 2.1 和之后的版本已经包含了 `unittest`。Python 2.0 用户则可以从 `pyunit.sourceforge.net`下载。

第 14 章 测试优先编程

- 14.3. roman.py, 第 3 阶段

> 注意
> 全面的单元测试能够告诉你的最重要的事情是什么时候停止编写代码。当一个函数的所有单元测试都通过了，停止编写这个函数。一旦整个模块的单元测试通过了，停止编写这个模块。

- 14.5. roman.py, 第 5 阶段

> 注意
> 当所有测试都通过了，停止编程。

第 15 章 重构

- 15.3. 重构

> 注意
> 在需要多次使用同一个正则表达式的情况下，应该将它进行编译以获得一个 pattern 对象，然后直接调用这个 pattern 对象的方法。

第 16 章 函数编程

- 16.2. 找到路径

> 注意
> 传递给 `os.path.abspath` 的路径名和文件名可以不存在。
>
> 注意
> `os.path.abspath` 不仅构建完整路径名，还能格式化路径名。这意味着如果你正工作于 `/usr/` 目录，`os.path.abspath('bin/../local/bin')` 将会返回 `/usr/local/bin`。它把路径名格式化为尽可能简单的形式。如果你只是希望简单地返回这样的格式化路径名而不需要完整路径名，可以使用 `os.path.normpath`。
>
> 注意
> 就像 `os` 和 `os.path` 模块的其他函数，`os.path.abspath` 是跨平台的。如果你是在 Windows (使用反斜杠作为路径符号) 或 Mac OS (使用冒号) 上运行，它们同样工作，只是将获得与我稍有不同的结果。`os` 的所有函数都是这样的。

第 17 章 动态函数

第 18 章 性能优化

- 18.2. 使用 timeit 模块

> 提示
> 你可以在命令行使用 `timeit` 模块来测试一个已存在的 Python 程序，而不需要修改代码。在 http://docs.python.org/lib/node396.html 查看文档中关于命令行选项的内容。
>
> 提示
> `timeit` 模块只有在你知道哪段代码需要优化时使用。如果你有一个很大的 Python 程序并且不知道你的性能问题所在，查看 `hotshot` 模块。

附录 D. 示例清单

第 1 章 安装 Python

- 1.3. Mac OS X 上的 Python
  - 例 1.1. 两个 Python 版本
- 1.5. RedHat Linux 上的 Python
  - 例 1.2. 在 RedHat Linux 9 上安装
- 1.6. Debian GNU/Linux 上的 Python
  - 例 1.3. 在 Debian GNU/Linux 上安装
- 1.7. 从源代码安装 Python
  - 例 1.4. 从源代码安装
- 1.8. 使用 Python 的交互 Shell
  - 例 1.5. 初次使用交互 Shell

第 2 章 第一个 Python 程序

- 2.1. 概览
  - 例 2.1. odbchelper.py
- 2.3. 文档化函数
  - 例 2.2. 定义 buildConnectionString 函数的 doc string
- 2.4. 万物皆对象
  - 例 2.3. 访问 buildConnectionString 函数的 doc string
- 2.4.1. 模块导入的搜索路径
  - 例 2.4. 模块导入的搜索路径
- 2.5. 代码缩进
  - 例 2.5. 缩进 buildConnectionString 函数
  - 例 2.6. if 语句

第 3 章 内置数据类型

- 3.1.1. Dictionary 的定义
  - 例 3.1. 定义 Dictionary
- 3.1.2. Dictionary 的修改
  - 例 3.2. 修改 Dictionary
  - 例 3.3. Dictionary 的 key 是大小写敏感的
  - 例 3.4. 在 dictionary 中混用数据类型
- 3.1.3. 从 dictionary 中删除元素
  - 例 3.5. 从 dictionary 中删除元素
- 3.2.1. List 的定义
  - 例 3.6. 定义 List

第 13 章 单元测试

第 14 章 测试优先编程

第 15 章 重构

第 17 章 动态函数

第 18 章 性能优化

附录 E. 修订历史

2007-09-01 (5.4b)

- XiaQ (xiaqqaix AT gmail DOT com) 进行了 Dive Into Python 中文版的修订工作，重发布为 5.4b 版。可以在 Dive Into Python 中文版的项目主页上找到一份完整的 (同时也是及其冗长的) 修订清单。注意原英文版不含 5.4b 版。

2006-04-25 (5.4 中文版)

参加了 Dive Into Python 中文版 5.4 版翻译的有：

- limodou (limodou AT gmail DOT com)
- Zoom.Quiet (zoom.quiet AT gmail DOT com)
- Jason Xie (jason AT sinoesl DOT org)
- YuLin (yulin724 AT gmail DOT com)
- Weigang LI (dimens AT gmail DOT com)
- wolfg (wolfg1969 AT gmail DOT com)
- Osmond (sinosmond AT gmail DOT com)

2004-05-20 (5.4)

- 添加了 第 12.1 节 “概览”。
- 添加了 第 12.2 节 “安装 SOAP 库”。
- 添加了 第 12.3 节 “步入 SOAP”。
- 添加了 第 12.4 节 “ SOAP 网络服务查错”。
- 添加了 第 12.5 节 “WSDL 介绍”。
- 添加了 第 12.6 节 “以 WSDL 进行 SOAP 内省”。
- 添加了 第 12.7 节 “搜索 Google”。
- 添加了 第 12.8 节 “ SOAP 网络服务故障排除”。
- 添加了 第 12.9 节 “小结”。
- 修改了 第 16 章 函数编程 和 第 18 章 性能优化 中的大量技术评论。

2004-05-12 (5.3)

- 为 第 18.3 节 “优化正则表达式” 添加了 `isalpha()` 的示例。感谢你，Paul。
- 对 第 5 章 对象和面向对象 和 第 6 章 异常和文件处理 进行了大量修改。

- 修正了 第 9.7 节 “Segue ” 中的 URL 地址。

2004-05-09 (5.2)

- 修正了 第 14.1 节 “roman.py, 第 1 阶段” 中的一个 URL 地址。
- 添加了 第 18.1 节 “概览”。
- 添加了 第 18.2 节 “使用 timeit 模块”。
- 添加了 第 18.3 节 “优化正则表达式”。
- 添加了 第 18.4 节 “优化字典查找”。
- 添加了 第 18.5 节 “优化列表操作”。
- 添加了 第 18.6 节 “优化字符串操作”。
- 添加了 第 18.7 节 “小结”。

2004-05-05 (5.1)

- 修订了 例 7.7 “校验十位数” 和 例 7.8 “用 {n,m} 语法确认罗马数字”。
- 修订了 例 7.10 “发现数字” 。
- 修正了 例 11.6 “测试 Last-Modified” 中的拼写错误。感谢你，Jesir。
- 修正了 例 3.11 “extend (扩展) 与 append (追加) 的差别” 与 append (追加) 的差别") 中的拼写错误。感谢你，Daniel。
- 修改了大量技术评论。
- 对 第 1 章 安装 *Python*、第 2 章 第一个 *Python* 程序、第 3 章 内置数据类型 和 第 4 章 自省的威力 进行了大量修改。

2004-04-16 (5.0)

- 添加了 第 11.1 节 “概览”.
- 添加了 第 11.2 节 “避免通过 HTTP 重复地获取数据”.
- 添加了 第 11.3 节 “HTTP 的特性”.
- 添加了 第 11.4 节 “调试 HTTP web 服务”.
- 添加了 第 11.5 节 “设置 User-Agent”.
- 添加了 第 11.6 节 “处理 Last-Modified 和 ETag”.
- 添加了 第 11.7 节 “处理重定向”.
- 添加了 第 11.8 节 “处理压缩数据”.
- 添加了 第 11.9 节 “全部放在一起”.
- 添加了 第 11.10 节 “小结”.
- 添加了 例 3.11 “extend (扩展) 与 append (追加) 的差别” 与 append (追加) 的差别").
- 在 第 1 章 安装 *Python* 中修改了关于如何下载 Python 的描述，使其更一般化，且减少了版本依赖性。

- 在 第 2.1 节 “概览” 和 第 2.4 节 “万物皆对象” 中，把术语 “module” (模块) 改为了 “program” (程序)，因为我们此时尚未介绍模块。
- 在 第 2.4 节 “万物皆对象” 中，添加了打开 Python IDE 的详细说明，和原来的例子合并在一起。
- 把所有引用到布尔值 `1` (真) 和 `0` (假) 分别改为了 `True` 和 `False` 。
- 更新了 例 3.22 “字符串格式化与字符串连接的比较”， 以展示 Python 2.3 所显示的 `TypeError` 消息。
- 修正了 例 17.19 “使用生成器替代递归” 中的拼写错误。
- 修正了 第 7.7 节 “小结” 中的拼写错误。
- 修正了 例 17.9 “plural4.py” 中的拼写错误。

2004-03-25 (4.9)

- 完成了 第 16.7 节 “全部放在一起”。
- 添加了 第 16.8 节 “小结”。
- 把对单元测试的介绍分成了两章： 第 13 章 单元测试 和 第 14 章 测试优先编程。
- 修正了 例 17.12 “plural4.py 的完成” 中的拼写错误。
- 修正了 例 17.18 “介绍生成器” 中的拼写错误。

2004-03-25 (4.8)

- 完成了 第 17.7 节 “plural.py, 第 6 阶段”。
- 完成了 第 17.8 节 “小结”。
- 修复了 附录 A, 进一步阅读、附录 B, 五分钟回顾、附录 C, 技巧和窍门和附录 D, 示例清单中的失效链接。

2004-03-21 (4.7)

- 添加了 第 17.1 节 “概览”。
- 添加了 第 17.2 节 “plural.py, 第 1 阶段”。
- 添加了 第 17.3 节 “plural.py, 第 2 阶段”。
- 添加了 第 17.4 节 “plural.py, 第 3 阶段”。
- 添加了 第 17.5 节 “plural.py, 第 4 阶段”。
- 添加了 第 17.6 节 “plural.py, 第 5 阶段”。
- 添加了 第 17.7 节 “plural.py, 第 6 阶段” (未完成)。
- 添加了 第 17.8 节 “小结” (未完成)。

2004-03-14 (4.6)

- 完成了 第 7.4 节 “使用 {n,m} 语法”。
- 完成了 第 7.5 节 “松散正则表达式”。
- 完成了 第 7.6 节 “个案研究：解析电话号码”。
- 扩写了 第 7.7 节 “小结”。

2004-03-07 (4.5)

- 添加了 第 7.1 节 “概览”。
- 添加了 第 7.4 节 “使用 {n,m} 语法” (未完成)。
- 添加了 第 7.5 节 “松散正则表达式” (未完成)。
- 添加了 第 7.6 节 “个案研究：解析电话号码” (未完成)。
- 添加了 第 7.7 节 “小结”。
- 把 第 7.2 节 “个案研究：街道地址” 和 第 7.3 节 “个案研究：罗马字母” 移入了正则表达式这一章。
- 添加了 例 6.20 “使用 glob 列出目录”。
- 添加了 例 6.7 “写入文件”。
- 添加了 例 5.11 “直接继承自内建数据类型 dict”。
- 添加了 例 10.11 “打印到 stderr”。
- 添加了 例 4.12 “使用getattr 创建分发者” 和 例 4.13 “getattr 缺省值”。
- 添加了 例 2.6 “if 语句”。
- 添加了 例 3.23 “数值的格式化”。
- 把 第 5 章 对象和面向对象 分成了两章：第 5 章 对象和面向对象 和 第 6 章 异常和文件处理。
- 把 第 9 章 *XML 处理* 分成了两章：第 9 章 *XML 处理* 和 第 10 章 脚本和流。
- 把 第 13 章 单元测试 分成了两章：第 13 章 单元测试 和 第 15 章 重构。
- 把 第 4 章 自省的威力 中的 `help` 改名为 `info`。
- 修正了 第 8.5 节 “locals 和 globals” 中的回向引用。
- 修复了 第 8.1 节 “概览” 中失效的示例链接。
- 修复了 第 9.1 节 “概览” 示例中丢失的若干行。
- 修正了 第 8.2 节 “sgmllib.py 介绍” 中的拼写错误。

2003-10-08 (4.4)

- 添加了 第 1.1 节 “哪一种 Python 适合您？”。
- 添加了 第 1.2 节 “Windows 上的 Python”。
- 添加了 第 1.3 节 “Mac OS X 上的 Python ”。
- 添加了 第 1.4 节 “Mac OS 9 上的 Python ”。
- 添加了 第 1.5 节 “RedHat Linux 上的 Python ”。
- 添加了 第 1.6 节 “Debian GNU/Linux 上的 Python ”。

- 添加了 第 1.7 节 “从源代码安装 Python ”。
- 添加了 第 1.9 节 “小结”。
- 删除了序言。
- 修正了 例 3.27 “odbchelper.py 的输出结果” 中的拼写错误。
- 为 第 2.3 节 “文档化函数” 添加了指向 PEP 257的链接。
- 修正了 第 3.4.2 节 “一次赋多值” 中指向 *How to Think Like a Computer Scientist* 的链接。
- 添加了 第 3.3 节 “Tuple 介绍” 中关于隐式断言 (assert) 的评注。

2003-09-28 (4.3)

- 添加了 第 16.6 节 “动态导入模块”。
- 添加了 第 16.7 节 “全部放在一起” (未完成)。
- 修正了 附录 F, 关于本书 中的链接。

2003-09-17 (4.2.1)

- 修正了主页中的链接。
- 修复了语法高亮。

2003-09-12 (4.2)

- 修正了 第 16.4 节 “重识列表映射”、第 16.3 节 “重识列表过滤”、第 7.2 节 “个案研究：街道地址” 和 第 10.6 节 “处理命令行参数” 中的拼写错误。感谢你，Doug。
- 修正了 第 5.3 节 “类的定义” 中的外部链接。感谢你，Harry。
- 修改了 第 4.5 节 “过滤列表” 结尾的措辞。感谢你，Paul。
- 为 第 13.5 节 “负面测试 (Testing for failure)"") 添加了几句话，以强调我们正在给 `assertRaises` 传递一个函数，而非函数名称。感谢你，Stephen。
- 修正了 第 8.8 节 “dialect.py 介绍” 中的拼写错误。感谢你，Wellie。
- 修正了关于方言示例的若干链接。
- 修正了关于罗马数字历史的外部链接。感谢世界各地的罗马数字爱好者。

2002-07-28 (4.1)

- 添加了 第 10.3 节 “查询缓冲节点”。
- 添加了 第 10.4 节 “查找节点的直接子节点”。
- 添加了 第 10.5 节 “根据节点类型创建不同的处理器”。
- 添加了 第 10.6 节 “处理命令行参数”。

- 添加了 第 10.7 节 “全部放在一起”。
- 添加了 第 10.8 节 “小结”。
- 修正了 第 6.5 节 “与目录共事” 中的拼写错误。应该是 `os.getcwd()` ，而不是 `os.path.getcwd()` 。感谢你，Abhishek。
- 修正了 第 3.7 节 “连接 list 与分割字符串” 中的排印错误。当在 Python IDE 中求值 (而非打印) 的时候，它会在输出的两侧加上单引号。
- 修改了 第 4.8 节 “全部放在一起” 中的 `str` 示例，以使用一个用户定义的函数，因为 Python 2.2 会为内置的字典模块定义一个 `doc string` ，从而导致原来那个示例的失效。感谢 Eric。
- 修正了 第 9.4 节 “Unicode” 中的拼写错误，“anyway”改为“anywhere”。感谢 Frank。
- 修正了 第 13.6 节 “完备性检测 (Testing for sanity)””) 中的拼写错误，我原把“accept”重复了一次。感谢 Ralph.
- 修正了 第 15.3 节 “重构” 中的排印错误， `C?C?C?` 匹配零至三个 `C` ，而不是四个。感谢 Ralph。
- 重写了 例 3.9 “Slice 简写” 中对分片的解说，使之更加清晰。感谢 Petr。
- 添加了 第 5.5 节 “探索 UserDict：一个封装类” 中的历史评注。现在 Python 2.2 支持直接子类化内置数据类型了。
- 为 例 5.9 “定义 UserDict 类” 添加了对字典的方法 `update` 的解说。感谢 Petr。
- 在 第 5.5 节 “探索 UserDict：一个封装类” 中阐明了 Python 对重载的不支持。感谢 Petr。
- 修正了 例 8.8 “BaseHTMLProcessor 介绍” 中的排印错误。HTML 注释以两个 (而非一个) 横杠和一个括号结束。感谢 Petr。
- 修改了 第 8.5 节 “locals 和 globals” 中关于嵌套作用域的注记。Python 2.2 已经发布了。感谢 Petr。
- 修正了 例 8.14 “BaseHTMLProcessor.py 中的基于 dictionary 的字符串格式化”; 中的排印错误：本应该是一连串空格，我误打为一个了。感谢 Petr。
- 添加了 第 5.5 节 “探索 UserDict：一个封装类” 的标题以标志基类。感谢 Petr。
- 添加了 第 15.3 节 “重构” 中的标题以提示下载 `unittest` 。感谢 Petr。
- 修正了 例 16.6 “在当前目录运行脚本” 中的排版错误；空格被误打为了制表符；行号也没有对准。感谢 Petr。
- 修正了 第 3.2 节 “List 介绍” 中首字母未大写的错误。应该是 `True` 和 `False` ，而非 `true` 和 `false` 。感谢所有指出这一点的读者。
- 修改了 第 3.1 节 “Dictionary 介绍”、第 3.2 节 “List 介绍” 和 第 3.3 节 “Tuple 介绍” 的小节标题。“Dictionaries 101”表明这是 Dictionary 的入门教程。美国的大学往往用“101”标示无需知识背景的入门课程，但显然这仅仅是个美国传统，所以没必要让我的国际读者们费劲猜测。——当我一年半前开始写这本书时，从未想到会拥有国际读者。
- 把样式表升级到 DocBook XSL 1.52 版。
- 升级到 由 Michael Kay 编制的 SAXON XSLT 解析器 6.52 版。
- 调整了样式表的相关部分。
- 上次修订后，她答应了。婚礼明年春天举行。

2002-04-26 (4.0-2)

- 修正了 例 4.15 “and 介绍” 中的拼写错误。
- 修正了 例 2.4 “模块导入的搜索路径” 中的拼写错误。
- 修复了 Windows 帮助文件 (样式表的改动引起了目录的丢失)。

2002-04-19 (4.0)

- 扩写了 第 2.4 节 “万物皆对象”，详细介绍了导入搜索路径。
- 修正了 例 3.7 “负的 list 索引” 中的拼写错误。感谢 Brian 的校正。
- 重写了 第 3.2 节 “List 介绍” 中关于真值的提示。现在 Python 有单独的布尔类型了。
- 修正了 第 5.2 节 “使用 from module import 导入模块” 中与 Java 的语法比较部分的拼写错误。感谢 Rick 的校正。
- 添加了 第 5.5 节 “探索 UserDict：一个封装类” 中的注解，说明了子类总是重写基类。
- 修正了 例 5.18 “修改类属性” 中的拼写错误。感谢 Kevin 的校正。
- 添加了 第 6.1 节 “异常处理” 中的注解，说明了可以自己定义异常并触发它。感谢 Rony 的建议。
- 修正了 例 8.17 “处理特别标记” 中的拼写错误。感谢 Rick 的校正。
- 添加了 例 8.18 “SGMLParser” 中的注解，详细说明了返回值。感谢 Howard 的建议。
- 在 例 10.6 “openAnything” 中，添加了 `StringIO` 初始化式中的 `str` 函数。感谢 Ganesan 的主意。
- 添加了 第 13.3 节 “romantest.py 介绍” 中的链接，解释了为什么测试用例要放在另一个文件中。
- 修改了 第 16.2 节 “找到路径”，用 `os.path.dirname` 代替了 `os.path.split` 。感谢 Marc 的主意。
- 添加了代码示例 (`piglatin.py` 、 `parsephone.py` 和 `plural.py`)，为正则表达式一章作准备。
- 更新并扩充了主页上关于 Python 的链接。

2002-01-01 (3.9)

- 添加了 第 9.4 节 “Unicode”。
- 添加了 第 9.5 节 “搜索元素”。
- 添加了 第 9.6 节 “访问元素属性”。
- 添加了 第 10.1 节 “抽象输入源”。
- 添加了 第 10.2 节 “标准输入、输出和错误”。
- 在 第 6.3 节 “for 循环” 中，为 `for` 循环示例添加了简单计数的例子 (好的用法和不好的用法)。感谢 Kevin 的主意。
- 修正了 例 3.25 “keys, values 和 items 函数” 中的排印错误 (`params.values()` 的两个元

素被颠倒了)。
- 修正了 第 4.3 节 “使用 type、str、dir 和其它内置函数” 中关于 `__builtin__` 模块的错误。感谢 Denis 的校正。
- 为 第 16.2 节 “找到路径” 添加了更多的示例，展示了如何在当前目录下运行单元测试，而不必进入 `regression.py` 所在的目录。
- 在 例 3.7 “负的 list 索引” 中，修改了如何由正索引得到负索引的解释。感谢 Renauld 的建议。
- 更新了主页上 Python 的下载链接。
- 为主页添加了指向 Bruce Eckel 的 *Thinking in Python* 初稿的链接。这是一本不可思议的 (也是高阶的) 书，讲述了设计模式以及如何用 Python 实现它们。

2001-11-18 (3.8)

- 添加了 第 16.2 节 “找到路径”。
- 添加了 第 16.3 节 “重识列表过滤”。
- 添加了 第 16.4 节 “重识列表映射”。
- 添加了 第 16.5 节 “数据中心思想编程”。
- 扩充了 第 16.1 节 “概览” 中的样例输出。
- 完成了 第 9.3 节 “XML 解析”。

2001-09-30 (3.7)

- 添加了 第 9.2 节 “包”。
- 添加了 第 9.3 节 “XML 解析”。
- 把 第 9.1 节 “概览” “打扫干净”了。感谢 Matt 的建议。
- 在 第 5.2 节 “使用 from module import 导入模块” 中，添加了关于 Java 的提示。感谢 Ori 的建议。
- 修正了 第 4.8 节 “全部放在一起” 中的错误。我曾暗示过，在 Python 中你不能使用 `is None` 来判断变量是否为 null。实际上可以；而且它比 `== None` 快。感谢 Ori 指出错误。
- 澄清了 第 3.2 节 “List 介绍”：我曾说 `li = li + other` 与 `li.extend(other)` 等价。操作结果的确相同；但 `extend` 更快，因为它不会创建新 list。感谢 Denis 指出。
- 修正了 第 3.2 节 “List 介绍” 中的错误：我曾说 `li += other` 与 `li = li + other` 等价。实际上，它与 `li.extend(other)` 等价，因为它不会创建新 list。感谢 Denis 指出。
- 修正了 第 2 章 第一个 *Python* 程序 因我的懒惰而引起的排版错误。当我写这一章时，我没有在字符串字面量的两侧加单引号。它们会被赋予特殊字体，但在这本书的一些编译版本 (如纯文本) 中这不奏效。感谢 Denis 的建议。
- 修正了 第 2.2 节 “函数声明” 中的错误。我曾说静态类型语言总是显示地要求给出变量的数据类型；事实上并非总是如此。有些静态类型语言会在编译时根据代码确定变量类

型。感谢 Tony 指出。
- 添加了指向 西班牙文版 的链接。

2001-09-06 (3.6.4)

- 为 `BaseHTMLProcessor` 添加处理非 HTML 实体引用的代码，并在 第 8.4 节 “BaseHTMLProcessor.py 介绍” 中添加了关于它的注解。
- 修改了 例 8.11 “globals 介绍”，使之在输出中包括 `htmlentitydefs` 。

2001-09-04 (3.6.3)

- 修正了 第 9.1 节 “概览” 中的拼写错误。
- 添加了指向 韩文版 的链接。

2001-08-31 (3.6.2)

- 修正了 第 13.6 节 “完备性检测 (Testing for sanity)”") 中的排印错误 (最后一点要求列了两次)。

2001-08-31 (3.6)

- 完成了 第 8 章 *HTML 处理*；在其末尾添加了 第 8.9 节 “全部放在一起” 和 第 8.10 节 “小结”。
- 添加了 第 15.4 节 “后记”。
- 在 第 9 章 *XML 处理* 开头添加了 第 9.1 节 “概览”。
- 在 第 16 章 函数编程 开头添加了 第 16.1 节 “概览”。
- 修正了一个长期存在的 bug：第 8 章 *HTML 处理* 中的示例被错误地着色了。
- 添加了指向 法文版 的链接。他们做对了：他们翻译了 XML 源文件，这样我的构建脚本得以重用，他们的成果就可以用六种格式发布。
- 升级到 DocBook XSL 1.43 版的样式表。
- 升级到 由 Michael Kay 编制的 SAXON XSLT 解析器 6.43 版。
- 对样式表进行了大规模翻修：放弃了原来的基于表格的排版，转向层叠样式表。不幸的是， CSS 总是会出兼容性问题，所以在每个页面的首尾部分都保留的表格。编译后的 HTML 版本在 Netscape 4 看起来更糟糕了；但在大多数现代浏览器中——包括 Netscape 6、Mozilla、Internet Explorer 5、Opera 5、Konqueror 和 iCab——中表现良好。在 Lynx 中完全可读。我爱 Lynx！它是我使用的第一个浏览器。你永远不会忘记你的第一次。
- 迁移到 Ant 平台上，以便控制进程——我现在可是在六种格式和两种语言中探险。

- 整理了所有的下载档案；这之前我为每个平台都发布了不同的文件，因为 Python 的 `zipfile` 模块创建的 .zip 文件不标准，因此不能被 Mac OS 上的 Aladdin Expander 打开。但 Ant 创建的 .zip 完全标准，兼容性良好。好样的，Ant！
- 我们把 XML 源、XSL 样式表和相关的脚本和库放在了 SourceForge。对于喜欢冒险的人们，当然有 CVS 接入。
- 在与 GPL 调和的 Python 2.1.1 license 下重新发布了代码。感谢你，Guido；人们确实关心，而它确实要紧。

2001-06-26 (3.5)

- 在 第 2.2 节 “函数声明” 中添加了对 强/弱/静态/动态 数据类型的注解。
- 添加了 第 3.1 节 “Dictionary 介绍” 中关于大小写敏感的示例。
- 在 第 5 章 对象和面向对象 中使用了 `os.path.normcase`，以兼容对大小写不敏感的糟糕的操作系统。
- 修正了 PDF 版本中代码的缩进问题。

2001-05-31 (3.4)

- 添加了 第 14.5 节 “roman.py, 第 5 阶段”。
- 添加了 第 15.1 节 “处理 bugs”。
- 添加了 第 15.2 节 “应对需求变化”。
- 添加了 第 15.3 节 “重构”。
- 添加了 第 15.5 节 “小结”。
- 又修正一个样式表 bug ：嵌套的 `</span>` 标记总是丢失。

2001-05-24 (3.3)

- 添加了 第 13.2 节 “深入”。
- 添加了 第 13.3 节 “romantest.py 介绍”。
- 添加了 第 13.4 节 “正面测试 (Testing for success)””)。
- 添加了 第 13.5 节 “负面测试 (Testing for failure)””)。
- 添加了 第 13.6 节 “完备性检测 (Testing for sanity)””)。
- 添加了 第 14.1 节 “roman.py, 第 1 阶段”。
- 添加了 第 14.2 节 “roman.py, 第 2 阶段”。
- 添加了 第 14.3 节 “roman.py, 第 3 阶段”。
- 添加了 第 14.4 节 “roman.py, 第 4 阶段”。
- 在兼容 Netscape/Mozilla 的强烈要求下，调整了样式表。

2001-05-03 (3.2)

- 添加了 第 8.8 节 “dialect.py 介绍”。
- 添加了 第 7.2 节 “个案研究：街道地址”。
- 修正了 `handle_decl` 方法中的 bug：它会产生不正确的声明 (把空格加在的不该出现的地方)。
- 修正了 CSS (2.9 介绍过) 中的 bug：背景色丢失了。

2001-04-18 (3.1)

- 为 `BaseHTMLProcessor.py` 添加处理声明的代码；现在 Python 2.1 也支持这段代码了。
- 添加 第 8.5 节 “locals 和 globals” 中关于嵌套作用域的注解。
- 修正了 例 8.1 “BaseHTMLProcessor.py” 中的一个隐秘的 bug：带有字符实体的属性值不会被合适地转义。
- 对于本书中的示例，现在推荐 (并不要求) Python 2.1，因为它支持 `sgmllib.py` 中的声明。
- 更新了 主页 上的链接，使其指向已经发布的 Python 2.1。
- 给文件加上了时间戳，以帮助重发布这本书的人们。

2001-04-16 (3.0)

- 对 第 8 章 *HTML* 处理 中的代码修正了几个小错误。
- 在 主页 上添加了指向中文版的链接。

2001-04-13 (2.9)

- 添加了 第 8.5 节 “locals 和 globals”。
- 添加了 第 8.6 节 “基于 dictionary 的字符串格式化”。
- 紧缩了 第 8 章 *HTML* 处理 中的代码，特别是 `ChefDialectizer` ，以使用更少、更简单的正则表达式。
- 修正了一个样式表 bug：它使得生成 PDF 版本时插入了不必要的空白页。
- 修正了一个脚本 bug：它剥除了 主页 上的 `DOCTYPE` 。
- 添加了指向 Python Cookbook](http://www.activestate.com/ASPN/Python/Cookbook/ "growing archive of annotated code samples") 的链接，并为 [附录 A, 进一步阅读 添加了一些链接。
- 改用 Google 代理 `http://diveintopython.org/` 的站内搜索。
- 升级到 DocBook XSL 样式表 1.36 版——比它所宣称的麻烦多了。可能还有潜伏的 bug。

2001-03-26 (2.8)

- 添加了 第 8.3 节 “从 HTML 文档中提取数据”。
- 添加了 第 8.4 节 “BaseHTMLProcessor.py 介绍”。
- 添加了 第 8.7 节 “给属性值加引号”。
- 紧缩了 第 4 章 自省的威力 中的代码：使用的内置函数 `callable` 来代替手工检测。
- 把 第 5.2 节 “使用 from module import 导入模块” 从 第 4 章 自省的威力 移到了 第 5 章 对象和面向对象。
- 修正了 第 5.1 节 “概览” 中的排印错误 (补回了冒号)。
- 添加了几个额外的供下载的样例脚本。
- 添加了 Windows Help 输出格式。

2001-03-16 (2.7)

- 添加了 第 8.2 节 “sgmllib.py 介绍”。
- 紧缩了 第 8 章 *HTML* 处理 中的代码。
- 修改了 第 2 章 第一个 *Python* 程序 中的代码：用了 `items` 方法代替了 `keys` 。
- 把 第 3.4.2 节 “一次赋多值” 一节移到了 第 2 章 第一个 *Python* 程序。
- 编辑了关于字符串 `join` 方法的评注，提供了 *The Whole Python FAQ* 中解释why `join` is a string method (为什么 `join` 是字符串方法) 而不是 list 方法的这一新条目。
- 重写了 第 4.6 节 “and 和 or 的特殊性质”，强调了 `and` 和 `or` 的基本性质；并且不再强调 `and-or` 技巧。
- 把语言比较重新组织为了 `note` 。

2001-02-28 (2.6)

- 给 PDF 和 Word 版本加入了彩色的示例文本、改进后的目录，以及缩进良好的 `tip` 和 `note` 。
- 现在 Word 版本是正宗的 Word 格式了，与 Word 97 完全兼容。
- 部分地解决 PDF 和纯文本版本中的特殊字符 (像 TM 符号和花括号) 错误的问题。
- 添加了指向支持 UNIX 的 Word 版本的下载链接——有些小鬼想把它导入到 StarOffice 或其它什么环境中去。
- 修正了几个丢失标题的 `注记` 。
- 修正了一个 bug：它会导致 Internet Explorer 5 for Mac OS 示例中的彩色文字发生字体错乱。(你好？！？Microsoft？好，你倒说说看，到底是 `<pre>` 的哪一部分你不懂？)
- 修复了 Mac OS 档案的下载链接。
- 在每章的第一个示例程序前，都添加了下载示例程序的链接。(我的日志显示人们往往会跳过本可以下载它们的那两个页面 (即：主页 和序言)，然后在阅读时惊慌失措地寻找下

载地址。)

- 进一步紧缩了 主页 和序言——但愿人们不会再因为其杂乱不堪而跳过它们。
- 但愿早点结束这没完没了的调试，回到写作正轨。

2001-02-23 (2.5)

- 添加了 第 6.4 节 “使用 sys.modules”。
- 添加了 第 6.5 节 “与目录共事”。
- 把 例 6.17 “分割路径名” 从 第 3.4.2 节 “一次赋多值” 移到了 第 6.5 节 “与目录共事”。
- 添加了 第 6.6 节 “全部放在一起”。
- 添加了 第 5.10 节 “小结”。
- 添加了 第 8.1 节 “概览”。
- 修正了 例 6.10 “遍历 dictionary” 中的程序清单——少了一个冒号。

2001-02-12 (2.4.1)

- 修改了新闻组链接，使用了 “news:” 协议。 `deja.com` 已死。
- 在下载链接旁标明了文件大小。

2001-02-12 (2.4)

- 为大多数章节添加了 “进一步阅读”，并在 附录 A, 进一步阅读 中汇总。
- 在纯文本格式的外部链接旁添加了 URL 地址。

2001-02-09 (2.3)

- 重写了 第 5 章 对象和面向对象 中的部分代码，使用了类的属性，并给出了一个更好的多值赋值的例子。
- 重新组织了 第 5 章 对象和面向对象，把类的部分提前了。
- 添加了 第 5.8 节 “类属性介绍”。
- 添加了 第 6.1 节 “异常处理”。
- 添加了 第 6.2 节 “与文件对象共事”。
- 把 第 5 章 对象和面向对象 中的 “回顾” 部分合并到了 第 5.1 节 “概览” 中。
- 给所有的程序清单和示例选择了配色方案。
- 修正了 第 2.2 节 “函数声明” 中的重大错误：函数会隐式地返回一个 `None` ，所以你 可以在不触发异常的前提下把它的返回值赋给一个变量。
- 扩写了 第 2.3 节 “文档化函数”、第 2.4 节 “万物皆对象” 和 第 3.4 节 “变量声明”，使之更加清晰。

2001-02-02 (2.2)

- 编辑了 第 4.4 节 “通过 getattr 获取对象引用”。
- 为 `xref` 标记添加了标题，所以它们也能显示可爱的提示框了。
- 修改了修订历史页面的主题风格。
- 修正了我昨天在我的 HTML 中提到的 bug：后期处理脚本将会导致无效的 HTML 字符引用，甚至让一些浏览器无法显示。
- 升级到 DocBook XSL 样式表 1.29 版。

2001-02-01 (2.1)

- 重写了 第 4 章 自省的威力 中的示例代码，用 `getattr` 代替了 `exec` 和 `eval`，并重写了相应的解释文字。
- 在 第 3.2 节 “List 介绍” 中添加了列表操作的示例。
- 在章节末尾的小节部分 (第 3.8 节 “小结” 和 第 4.9 节 “小结”) 添加了指向相关内容的链接。

2001-01-31 (2.0)

- 把 第 5.6 节 “专用类方法” 分成了三节： 第 5.5 节 “探索 UserDict：一个封装类”、第 5.6 节 “专用类方法” 和 第 5.7 节 “高级专用类方法”。
- 修改了关于 垃圾收集 的注解以指出 Python 2.0 及其后续版本自动处理循环引用，无需手工编码。
- 修复了 UNIX 下载，使之包括了所有相关文件。

2001-01-15 (1.9)

- 删除了 第 2 章 第一个 *Python* 程序 的概论。
- 删除了 第 4 章 自省的威力 的概论。
- 删除了 第 5 章 对象和面向对象 的概论。
- 大刀阔斧地改动文本。我似乎总是跑题。

2001-01-12 (1.8)

- 为 第 3.4.2 节 “一次赋多值” 添加了更多示例。
- 添加了 第 5.3 节 “类的定义”。
- 添加了 第 5.4 节 “类的实例化”。
- 添加了 第 5.6 节 “专用类方法”。

- 一些小的样式表调整，包括为 `link` 标记添加了标题。要是你的浏览器够酷的话，它会用一个可爱的提示框显示链接目标的描述。

2001-01-03 (1.71)

- 对样式表进行了几处修改，提高了对浏览器的兼容性。

2001-01-02 (1.7)

- 添加了 第 2 章 第一个 *Python* 程序 的概览。
- 添加了 第 4 章 自省的威力 的概览。
- 添加了 第 5 章 对象和面向对象 的小结部分[后来删除了]。
- 添加了 第 5.9 节 “私有函数”。
- 添加了 第 6.3 节 “for 循环”。
- 添加了 第 3.4.2 节 “一次赋多值”。
- 编写了把书输出为单个 HTML 文件、PDF、Microsoft Word 97 和纯文本的脚本。
- 注册了域名 `diveintopython.org` 并把书——在线的和离线的——都移到了那里。
- 修改了 XSL 样式表，以修改在每个页面首尾显示的导航链接。页首加上了域名、版本号，以及跳到章目录、总目录和主页的快速链接。

2000-12-11 (1.6)

- 添加了 第 4.8 节 “全部放在一起”。
- 完成了 第 4 章 自省的威力：在结尾添加了 第 4.9 节 “小结”。
- 在 第 5 章 对象和面向对象 的开头添加了 第 5.1 节 “概览”。

2000-11-22 (1.5)

- 添加了 第 4.6 节 “and 和 or 的特殊性质”。
- 添加了 第 4.7 节 “使用 lambda 函数”。
- 添加了附录 五分钟回顾。
- 添加了附录 技巧和窍门。
- 添加了附录 示例清单。
- 添加了附录 修订历史。
- 扩写了 第 3.6 节 “映射 list” 中映射列表的示例。
- 把更多的常用术语加入了 XML 实体。
- 升级到 DocBook XSL 样式表 1.25 版。

2000-11-14 (1.4)

- 添加了 第 4.5 节 “过滤列表”。
- 为 第 4.3 节 “使用 type、str、dir 和其它内置函数” 添加了 `dir` 文档。
- 在 第 3.3 节 “Tuple 介绍” 中，添加了 `in` 的示例。
- 添加了 MacPython 环境下 `if` `__name__` 技巧的额外说明。
- 转向了 由 Michael Kay 编制的 SAXON XSLT 解析器。
- 升级到 DocBook XSL 样式表 1.24 版。
- 添加了 db-html 处理的说明：在每个章、节中都加上了显示的文件名，避免在添加或重排节时导致深层链接实效。
- 把一些常用术语加入了 XML 实体。
- 把几个 `literal` 标记改为了 `constant` 。

2000-11-09 (1.3)

- 添加了关于动态代码执行的部分。
- 在所有有必要的地方，都添加了对本书其它部分的引用链接。
- 扩写了 第 2 章概览，解释了这个函数的实际作用。
- 显示地声明示例代码是在 GNU General Public License 下发布的；添加了该 License 的附录。 [Note 8/16/2001：代码在与 GPL 调和的 Python license 下重新发布了。]
- 修改了链接，以使用 `xref` 标记——我弄明白怎么使用了这个标记了。

2000-11-06 (1.2)

- 添加了第 2 章的前四部分。
- 进一步紧缩了了序言，并添加了指向 Python 的 Mac OS 版本的链接。
- 扩写了“映射列表”和“连接字符串”中的示例，使其更富逻辑性。
- 添加了第 1 章小结中的输出。

2000-10-31 (1.1)

- 完成了第 1 章关于映射、连接的部分，以及整章小结。
- 在序言使用了较缓和的措辞，并添加了针对非程序员的入门教程的链接。
- 修正了几处拼写错误。

2000-10-30 (1.0)

- 原始发布

附录 F. 关于本书

本书的英文版是用 Emacs 以 DocBook XML 格式写成的, 使用 由 Norman Walsh 编制的 XSL 样式表 的经过客户化的版本由 由 Michael Kay 编制的 SAXON XSLT 解析器 转化为 HTML。在此基础上, 使用 HTMLDoc 将其转化为 PDF; 使用 w3m 将其转化为无格式文本 (plain text)。(本书的简体中文版是使用译者习惯使用的编辑器对 XML 源代码进行翻译, 并使用原书作者提供的创建脚本输出了 html 版和 htmlflat 版, 使用 HTML Help Workshop 生成了chm版本。)程序清单和例子使用了由 Just van Rossum 编写的 `pyfontify.py` 程序的改进版本进行了语法着色, 此程序包含在示例脚本中。

如果您对学习用 DocBook 进行技术方面的写作感兴趣, 可以 下载本书的 XML 源代码 和 创建脚本, 它包含了用于创建本书各种输出格式的经过客户化的 XSL 样式表。您还应该阅读权威著作, *DocBook: The Definitive Guide*。如果您打算用 DocBook 进行任何正式的写作, 我建议您订阅 DocBook 邮件列表。

附录 G. GNU Free Documentation License

Version 1.1, March 2000

G.0. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

G.1. Applicability and definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

G.2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.3. Copying in quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

G.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor

acknowledgements and/or dedications given therein.

12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

G.5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

G.6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

G.7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

G.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

G.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

G.10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

G.11. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

附录 H. GNU 自由文档协议

1.1 版, 2000 年 5 月

This is an unofficial translation of the GNU Free Documentation License (GFDL) into Chinese. It was not published by the Free Software Foundation, and does not legally state the distribution terms for works that uses the GFDL --only the original English text of the GFDL does that. However, I hope that this translation will help Chinese speakers understand the GFDL better.

这是 GNU 自由文档协议的非官方中文翻译。它不由自由软件基金会出版，因此不代表使用 GFDL 的著作的法律合同——只有 GFDL 的英文原文具有此效力。然而我们希望此翻译有助于说中文的人更好地理解 GFDL 。

H.0. 序

本协议的目的在于保证一本手册、教程，或其它任何书面文档在如下意义上的“自由”：确保任何人复制、重发行它的自由，不论是否经过修改，不论是否出于商业目的；其次，这一协议保护文档的作者和出版者的信誉不受他人修改的影响。

本协议是一种“copyleft”，这意味着：文档的派生物和文档本身必须遵循同种意义上的自由。本协议是对 GNU 通用公共协议的补充，后者是为自由软件撰写的“copyleft”协议。

我们撰写这一协议的最初目的是用于自由软件的手册，因为自由软件需要自由文档：自由软件应当带有同样自由的文档。但这一协议的适用范围不局限于软件文档；任何书面著作都可以使用本协议，不论它涉及哪一主题，不论它是否会出版为纸质书籍。我们建议那些以指导或参考为目的的文档作品使用本协议。

H.1. 适用范围和定义

本协议适用于任何手册或者其它著作，只要版权人声明了以本协议发行。下文所说的“文档”是指任何此类手册或著作。任何公众成员都是协议的受益者，我们将用“你”来称呼。

文档的“修改版”(Modified Version) 指任何包括文档或其一部分的著作，不论是原样复制的，还是加入了修改和/或翻译。

“附属部分” (Secondary Section) 指文档中指定的附录或序言部分，用于专门叙述作者或出版者与文档主题 (或其相关事项) 之间的关系，不包含该主题内的任何具体内容。(例如，如果文档是数学教材的一部分，那么附属部分可能不会讲述任何数学知识。)上面所说的关系可能是与文档主题 (或其相关事项) 之间的历史关联，或对它的法律、商业、哲学、伦理或政治观点。

“不变部分”(Invariant Sections) 是“附属部分”的一个特定子集，由使用本协议的声明根据其标题指定。

“封面文本”(Cover Texts) 是特定的短篇文字，在使用本协议的声明中列为“前封面文本”或“封底文本”。

文档的“透明”(Transparent) 副本指这样的副本：可以由机器阅读；其格式标准是公众可得的，其内容可以用一般的文本编辑器，或 (对于像素图象) 一般的绘图程序 ，或 (对于绘画作品) 广泛使用的图象编辑器直接打开查看和编辑；能直接输入文本格式器，或可自动化翻译为多种能输入文本格式器的格式。如果某一副本以非透明文件格式制作，后者的目的在于阻挠或不鼓励读者进行后期的修改，则该副本不是透明的，即“非透明”(Opaque) 的。

透明副本应当采用的合适格式的例子：不带标记的纯 ASCII 文本、Texinfo 输入格式、LaTeX 输入格式、采用公众可得的 DTD 的 SGML 或 XML，以及遵循标准的简单的为人类修改所设计的 HTML。非透明的格式包括 PostScript、PDF、只能被专有文字处理器阅读和编辑的专有格式、采用公众不可得的 DTD 和/或处理工具的 SGML 或 XML、机器生成的，仅被一些文字处理器用于输出的 HTML。

“扉页”(Title Page)指的是——对于纸版书来说——扉页本身，加上本协议要求出现在扉页中的材料所在的页面。对那些格式上没有此类扉页的作品，“扉页”指的是出现在作品最显著标题附近，正文前的文字。

H.2. 原样复制

你可以以任何媒介复制和发行文档——不论是否出于商业目的——只要在每个副本中包含本协议、版权声明和采用本协议的声明，同时不给本协议附加任何其它条件。你不能使用技术手段妨碍或控制你所制作或发行的副本的阅读和继续复制。然而，你可以为你的复制接受偿金。如果你发行了大量的副本，你还要遵循第 3 部分中的条件。

你也可以借出副本——条件和上面所说的相同——还可以公开展示。

H.3. 大量复制

如果你出版的印刷的文档副本数超过 100，而且文档的协议声明要求包括封面文本，你必须为每份副本都装上包含清楚易认的封面文本的封面：前封面文本放在前封面，封底文本放在封底。两个封面都必须清楚易认地指出你是这些副本的出版者。前封面必须印上完整的标题，每个单词都同样显著和易认。你可以在封面上添加其它内容。如果某一复制的改动仅限于封面，且保留了原有标题并符合上述条件，那么在其它方面就可以被认为是原样复制。

如果要求包括的封面文本 (前封面的或封底的) 过于冗长以致不能满足易认的属性，你应当把列在前面的内容 (数量尽量合适地) 放在封面上，余下的内容放在后续页面上。

如果你出版或发行的非透明文档的副本数超过 100，你必须或者为每个非透明副本附带一份可机读的透明副本；或者在每个非透明副本内，或附带副本指出一个公众可进入的计算机网络地址，后者包含了一份完整的不带附加材料的透明文档副本，一般的网络使用者使用公共标准的网络接口，无需付费即可匿名下载。如果你选用了后一种办法，你必须在开始发行非透明副本时采取一些谨慎的措施，以确保处在你所指明的地址的透明副本在你发行最后一本副本 (不论是直接发行的，还是通过代理商或零售商发行的) 后至少一年内仍然可用。

请求，但不强制要求，你在重发行大量副本之前联系文档的作者，以便他们能给你提供文档的更新版本。

H.4. 修改

满足了第 2 和第 3 部分中的要求，你就可以复制和发行文档的修改版，其前提是你精确地把修改版以本协议发布，此时修改版就扮演了文档的角色，本协议所规定的发行和修改条款就对修改版的持有者有效。此外你必须对修改版做到：

1. 在扉页 (以及封面，如果有的话) 上使用和文档不同的标题，同时也和上一版本不同 (上一版本应当被列在文档的历史部分中，如果有的话)。如果得到了上一版本的出版者的许可，你也可以使用同上版本相同的标题。
2. 在扉页上列出一个或多个为修改版的著作权负责的作者，并列出至少五个主要作者 (如果不足五个则列出所有作者)。
3. 在扉页上指出修改版的出版者，并以其作为书的出版者。
4. 保留文档的所有版权声明。
5. 在其它版权声明旁边，添加上合适的针对你的修改的版权声明。
6. 在版权声明和后面紧接着包括一段协议声明，允许以本协议使用修改版，其格式如附录中所示。
7. 完整地保留文档的协议声明中给出的不变部分和要求包括的封面文本。
8. 包含本协议的一份未经改动的副本。
9. 保留“历史”部分及其标题，并为其添加一项，至少包括扉页中所指出的标题、年份、新作者和出版者。如果文档中没有标题为“历史”的部分，创建一个，声明文档的扉页中所指出的标题、年份、作者和出版者，再添加一项，以描述修改版，正如上文所述。
10. 保留文档中给出的用以获取文档的透明副本的网络地址——如果有的话；同样包括修改版所基于的前一版本所给出的地址。这些都可以放在“历史”部分里面。如果著作是在文档本身至少四年前出版的，或者该版本的原出版者允许，你可以略去它的地址。
11. 保留原文档中的“鸣谢”或“贡献”部分的标题，以及其中给予每一个贡献者的鸣谢和/或献辞的要旨和语气。
12. 保留文档的所有不变部分，文字和标题均不能改动。章节号或其等价物不被视为章节标题的一部分。
13. 删除所有题为“赞誉”的章节。这样的章节不能包括在修改版中。
14. 不要把任何已有章节题为“赞誉”，或与不变部分中的标题冲突。

如果修改版包括新的序言或附录，且满足附属部分的条件，不包含从文档中复制的材料，你可以 (也可以不) 把它们的一部分或全体指定为不变的。要做到这一点，需要把它们的标题添加到修改版的协议声明的不变部分列表中。这些标题必须和其它章节标题不同。

你可以添加题为“赞誉”的一个章节，只要它只包括不同群体对你的修改版的赞誉——例如，同类作品的对比，或者你的文字已被某个组织或官方机构批准为某个标准的定义的原文。

你可以在修改版封面文本的最后添加一小段——最多五个单词——作为前封面文本，另一小段——最多 25 个单词——作为封底文本。一个实体只能添加 (或整理出) 前封面文本的一段和封底文本的一段。如果文档的某一封面已经包括了你或你所代表的同一个实体之前所添加 (或整理出) 的一段封面文本，你不能再往这一封面上添加；但你可以替换掉旧的文本，只要得到了添加该文本的出版者的明确许可。

文档的作者和出版者使用本协议，并不连同给予此许可：公开使用他们的名字来确保或暗示对修改版的肯定。

H.5. 合并文档

你可以把文档以及其它以本协议发布的文档合并起来，前提是符合第 4 部分中关于修改版的条件，并在合并作品中不经修改地包含所有原文档的所有不变部分，并在合并作品的协议声明中把它们全部列为不变部分。

合并作品只需包含本协议的一份副本，重复的不变部分可以合并为一个。如果几个不变部分的标题相同而内容不同，则应在标题的末尾添加其原作者或出版者的名字 (用括号括起来)——如果是已知的——来区分，否则就用一个独有的数字来区分。对合并作品的协议声明的不变部分列表中出现的章节标题作同样的调整。

合并时，你必须把各个原文档的“历史”部分合并为一个“历史”部分；对任何题为“鸣谢”和“献辞”的部分作同样的合并。你必须删除所有题为“赞誉”的部分。

H.6. 文档合集

你可以把文档以及其它以本协议发布的文档合为一集，把分立的本协议的副本合为一个包含在合集中，前提是对每个文档的其它所有方面，都应遵循本协议关于原样复制的规定。

你可以从这样的合集中抽取出一个文档，并以本协议单独发行，前提是你把本协议的副本插入到了抽取出的文档中，并在其它所有方面，都应遵循本协议关于原样复制的规定。

H.7. 独立著作聚集

文档或其派生物与其它分立文档或著作的汇编，以一个文库或一个大媒体的形式发行，如果其自身没有汇编版权声明，其整体不被视为文档的修改版。这样的汇编称为“聚集”，本协议不约束和文档汇编在一起的其它非派生作品。

如果第 3 部分中关于封面文本的要求对于这些 (聚集中的) 文档副本是可行的，但文档只占聚集的不到 1/4 的部分，那么文档的封面文本可以被置于聚集内部，只围绕在文档自身周围。否则，它们必须出现在聚集的封面上。

H.8. 翻译

翻译被认为是修改的一种，所以你可以依据第 4 分部分发行文档的翻译版。把不变部分替换为翻译版本需要得到版权持有者的特别许可，但你可以包括部分或全部不变部分的翻译版本作为补充。你也可以包括本协议的一份翻译，但同时也要包括本协议的英语版本。如果翻译版本和英语原版有任何出入，当以英语原版为准。

H.9. 终止协议

你只能在明确遵循本协议的前提下，对文档进行复制、修改、派生协议或发行。其它任何任何复制、修改、派生协议或发行的企图都是无效的，并会自动终止本协议授予你的权力。然而如果你违反了本协议，那些根据此协议从你处接受了副本或者权力的人们，只要遵循本协议，他们的协议将不会被终止。

H.10. 协议将来的修订

自由软件基金会可能会不时出版 GNU 自由文档协议的新的、修订过的版本。这样的新版本在其精神上会与这一版本相似，但为对付新的问题或利害关系，在细节上可能会有所不同。参看 http://www.gnu.org/copyleft/。

协议的每一版本都会被赋予一个以示区别的版本号。如果文档指定和本协议的版本号加上“或者更新的版本”，你可以遵循指定的版本和自由软件基金会出版的新版本 (不包括草稿) 中的任意一个。如果文档没有指定版本号，你可以从自由软件基金会曾经出版过的版本 (不包括草稿) 中任选一个。

H.11. 如何为你的文档使用本协议

要为你所写的文档使用本协议，在文档中包括本协议的一份副本，并把如下版权和协议声明放在扉页后：

> Copyright (c) YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".
>
> 参考译文：
>
> Copyright (c) YEAR YOUR NAME.
>
> 允许在 GNU 自由文档协议 (1.1 版，或自由软件基金会出版的任何更新版本) 的许可下复制、发行且/或修改本文档；本文档的不变部分包括[它们的标题列在这里]，前封面文本是[列在这里]，封底文本是[列在这里]。该协议的一份副本包含在题为“GNU Free Documentation License”的章节中。

如果没有不变部分，就写“with no Invariant Sections”(没有不变部分) 而不是指明不变部分；如果没有前封面文本，就写“no Front-Cover Texts”(没有前封面文本) 而不是“Front-Cover Texts being LIST”(前封面文本是[列在这里])；封底文本也是一样。

如果你的文档包含有价值的样例程序代码，我们建议你把这些样例用平行的自由软件协议发布，以使得它们可以在自由软件中被使用，例如 GNU 通用公共协议。

附录 I. Python license

I.A. History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of a language called ABC. Guido is Python's principal author, although it includes many contributions from others. The last version released from CWI was Python 1.2. In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia where he released several versions of the software. Python 1.6 was the last of the versions released by CNRI. In 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. Python 2.0 was the first and only release from BeOpen.com.

Following the release of Python 1.6, and after Guido van Rossum left CNRI to work with commercial software developers, it became clear that the ability to use Python with software available under the GNU Public License (GPL) was very desirable. CNRI and the Free Software Foundation (FSF) interacted to develop enabling wording changes to the Python license. Python 1.6.1 is essentially the same as Python 1.6, with a few minor bug fixes, and with a different license that enables later versions to be GPL-compatible. Python 2.1 is a derivative work of Python 1.6.1, as well as of Python 2.0.

After Python 2.0 was released by BeOpen.com, Guido van Rossum and the other PythonLabs developers joined Digital Creations. All intellectual property added from this point on, starting with Python 2.1 and its alpha and beta releases, is owned by the Python Software Foundation (PSF), a non-profit modeled after the Apache Software Foundation. See http://www.python.org/psf/ for more information about the PSF.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

I.B. Terms and conditions for accessing or otherwise using Python

I.B.1. PSF license agreement

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.1.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.1.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001 Python Software Foundation; All Rights Reserved" are retained in Python 2.1.1 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.1.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.1.1.
4. PSF is making Python 2.1.1 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.1.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.1.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.1.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of

agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.1.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

I.B.2. BeOpen Python open source license agreement version 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the

"BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

I.B.3. CNRI open source GPL-compatible license agreement

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright (c) 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013".
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY

THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

I.B.4. CWI permissions statement and disclaimer

> Copyright (c) 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

附录 J. Python 协议

J.0. 关于译文的声明

本译文仅供参考，不代表 Python 协议本身。它不具备任何法律效力——只有 Python 协议的英文原文具有此效力。

J.A. 软件的历史

Python 是 Guido van Rossum 于 20 世纪 90 年代早期在荷兰的 Stichting Mathematisch Centrum (CWI) 发明的，作为一门叫 ABC 的语言的后继。Guido 是 Python 的主要作者，尽管它包括了很多他人的贡献。CWI 发布的最后版本是 Python 1.2。1995 年，Guido 在 Reston,Virginia 的 Corporation for National Research Initiatives (CNRI) 继续进行 Python 的工作，他在此又发布了这个软件的多个版本。Python 1.6 是 CNRI 发布的最后版本。2000 年，Guido 和 Python 的核心团队转移到了 BeOpen.com，形成了 BeOpen PythonLabs 团队。Python 2.0 第一个也是唯一一个 BeOpen.com 发布的版本。

Python 1.6 发布后，Guido van Rossum 离开了 CNRI，同商业软件开发员一同工作。让 Python 和以 GNU Public License (GPL) 发布的软件一起工作的需求逐渐明朗起来。CNRI 和自由软件基金会 (Free Software Foundation, FSF) 进行了接触，得到对 Python 协议改动措辞的许可。Python 1.6.1 本质上与 Python 1.6 是相同的，只修正了小部分 bug，但却是用不同的协议发布，从而后续的版本都兼容 GPL 了。Python 2.1 是 Python 1.6.1 的衍生物，Python 2.0 也是。

Python 2.0 在 BeOpen.com 发布后，Guido van Rossum 和其它的 PythonLabs 开发者加入了 Digital Creations。从 Python 2.1 及其 alpha 和 beta 发布开始，此后加入的所有知识产权都由 Python 软件基金会 (Python Software Foundation, PSF) 所有，后者是一个非盈利的组织，以 Apache 软件基金会为样板。关于 PSF 的更多信息可以参看 http://www.python.org/psf/。

感谢成千上万的外部志愿者，他们在 Guido 的指引下使得所有这些发布成为可能。

J.B. 使用 Python 的条款和条件

J.B.1. PSF 协议

1. 本协议是 Python 软件基金会 (“PSF”) 同使用 Python 2.1.1 软件——源码或二进制形式——及其相关文档的个人或组织 (“获许者”) 之间达成的契约。
2. 在服从本协议的条款和条件的前提下，PSF 特此给予获许者无例外、无版税、全球的许可，以复制、分析、测试、运行和/或公开展出、衍生、发行，或以其它方式使用单独的 Python 2.1.1 或其衍生版本，前提是 PSF 的协议和 PSF 的版权声明 (“Copyright (c) 2001 Python Software Foundation; All Rights Reserved”) 保留在单独的 Python 2.1.1 或其由获许者特制的衍生版本中。
3. 如果获许者制作了一基于或包括 Python 2.1.1 或其一部分的衍生物，同时想要使其能为他人所得，本协议特此同意在其中包含关于对 Python 2.1.1 所作改动的简短概要。
4. PSF 使获许者能够使用 Python 2.1.1，其基础是“AS IS”原则。PSF 不作任何描述或保证，不论是明确的还是暗含的。例如——但决不限于此——PSF 不作并且否认如下描述或保证：产品的销路，产品是否适合于某一用途，以及 Python 2.1.1 不会侵犯任何第三方的权力。
5. PSF 将不对获许者或其它任何 Python 2.1.1 的用户，为修改、发行或者以其它方式使用 Python 2.1.1 或其衍生物所引起的任何意外的、特殊的，或直接的损害或丢失负担任何责任，即使对此种可能性知情。
6. 一旦协议的条款和条件遭到实质性的破坏，本协议将自动终止。
7. 协议中的任何内容都不视为要在 PSF 和获许者之间创造代理、合作或者合资的关系。本协议不授予此许可：使用 PSF 商标，或在商标中以商标的意义使用其商标名，以赞誉或宣扬获许者或任何第三方的产品或服务。
8. 通过复制、安装，或以其它方式使用 Python 2.1.1，获许者同意受到本协议的条款和条件的约束。

J.B.2. BeOpen Python 开源协议第 1 版

1. 本协议是 BeOpen.com (“BeOpen”，其办公室位于 160 Saratoga Avenue, Santa Clara, CA 95051) 同使用本软件——源码或二进制形式——及其相关文档 (“本软件”) 的个人或组织 (“获许者”) 之间达成的契约。
2. 在服从 BeOpen Python 协议的前提下，BeOpen 特此给予获许者无例外、无版税、全球

的许可，以复制、分析、测试、运行和/或公开展出、衍生、发行，或以其它方式使用单独的本软件或其衍生版本，前提是 BeOpen Python 保留在单独的本软件或其由获许者特制的衍生版本中。

3. BeOpen 使获许者能够使用本软件，其基础是“AS IS”原则。BeOpen 不作任何描述或保证，不论是明确的还是暗含的。例如——但决不限于此——BeOpen 不作并且否认如下描述或保证：产品的销路，产品是否适合于某一用途，以及本软件不会侵犯任何第三方的权力。
4. PSF 将不对获许者或其它任何本软件的用户，为修改、发行或者以其它方式使用本软件或其衍生物所引起的任何意外的、特殊的，或直接的损害或丢失负担任何责任，即使对此种可能性知情。
5. 一旦协议的条款和条件遭到实质性的破坏，本协议将自动终止。
6. 本协议应当受到 California 州法律的管理和解释，除非法律条款发生了矛盾。协议中的任何内容都不视为要在 BeOpen 和获许者之间创造代理、合作或者合资的关系。本协议不授予此许可：使用 BeOpen 商标，或在商标中以商标的意义使用其商标名，以赞誉或宣扬获许者或任何第三方的产品或服务。作为例外，出现在 http://www.pythonlabs.com/logos.html 的“BeOpen Python”标志可以按照该网页上所授予的许可使用。
7. 通过复制、安装，或以其它方式使用本软件，获许者同意受到本协议的条款和条件的约束。

J.B.3. CNRI 开源 GPL 兼容协议

1. 本协议是 Corporation for National Research Initiatives (“CNRI”，其办公室位于 1895 Preston White Drive, Reston, VA 20191) 同使用 Python 1.6.1——源码或二进制形式——及其相关文档的个人或组织之间达成的契约。
2. 在服从本协议的条款和条件的前提下，CNRI 特此给予获许者无例外、无版税、全球的许可，以复制、分析、测试、运行和/或公开展出、衍生、发行，或以其它方式使用单独的 Python 1.6.1 或其衍生版本，前提是 CNRI 的协议和 CNRI 的版权声明 (“Copyright (c) 1995-2001 Corporation for National Research Initiatives; All Rights Reserved”) 保留在单独的 Python 1.6.1 或其由获许者特制的衍生版本中。作为替代，在应当出现 CNRI 协议之处，获许者可以用下列文本加以替代：(不含引号)：“Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013”(Python 1.6.1 必须在服从 CNRI 的协议的前提下方可使用。这一协议和 Python 1.6.1 一起，用一个唯一的、不变的标识符 (也叫标识句柄) 即可在 Internet 上找到：1895.22/1013。这一协议可以通过代理服务器从 Internet 上得到，通过访问如下 URL：http://hdl.handle.net/1895.22/1013)。
3. 如果获许者制作了一基于或包括 Python 1.6.1 或其一部分的衍生物，同时想要使其能为

他人所得，本协议特此同意在其中包含关于对 Python 1.6.1 所作改动的简短概要。

4. CNRI 使获许者能够使用 1.6.1，其基础是“AS IS”原则。CNRI 不作任何描述或保证，不论是明确的还是暗含的。例如——但决不限于此——CNRI 不作并且否认如下描述或保证：产品的销路，产品是否适合于某一用途，以及 Python 1.6.1 不会侵犯任何第三方的权力。
5. CNRI 将不对获许者或其它任何 Python 1.6.1 的用户，为修改、发行或者以其它方式使用 Python 1.6.1 或其衍生物所引起的任何意外的、特殊的，或直接的损害或丢失负担任何责任，即使对此种可能性知情。
6. 一旦协议的条款和条件遭到实质性的破坏，本协议将自动终止。
7. 本协议应当受到美国联邦知识产权法律的管理，包括但不限于联邦版权法——或者，在联邦法律不起作用的地方，如 Virginia 州，由 Commonwealth of Virginia 制定的法律，除非其条款发生了矛盾。尽管如前所述，考虑到基于 Python 1.6.1 的包含不可分割的以 GNU 通用公共协议 (GPL) 发布的材料的衍生物，由 Commonwealth of Virginia 制定的法律只管理使用本协议时与第 4、5、7 章中的内容有关的事项。协议中的任何内容都不视为要在 CNRI 和获许者之间创造代理、合作或者合资的关系。本协议不授予此许可：使用 CNRI 商标，或在商标中以商标的意义使用其商标名，以赞誉或宣扬获许者或任何第三方的产品或服务。
8. 通过单击标示出“接受”的按钮、或者复制、安装，或以其它方式使用 Python 1.6.1，获许者同意受到本协议的条款和条件的约束。

J.B.4. CWI 许可声明与免责声明

在此特许以任何目的、不付费地使用、复制、修改和发行本软件及其文档，其前提是上述版权声明出现在所有的副本中，版权声明和此许可声明出现在支持文档中。在未经书面特许，Stichting Mathematisch Centrum 或 CWI 的名字不被用于广告或软件发行的相关宣传事宜。

Stichting Mathematisch Centrum 否认所有对此软件的保证，包括所有暗含的对产品的销路、是否适合于某一用途的保证。无论如何，Stichting Mathematisch Centrum 都不为任何意外的、间接的，或直接的损害或由任何由使用权、数据或利益的损失所造成的损害负担任何责任，不论是否达成了合同，还是由于疏忽大意或者其它的侵权行为，不论和使用或运行本软件是否有关。